最新

まるごと図解

C言語がわかる

高田美樹◉著



- プログラムは**どうして**動くのか、プログラミングとは**何を**するものなのか
- **アルゴリズム**とは何か
- プログラムの3つの基本構造 **順次・選択・反復**
- C言語は**構造化プログラミング**向き
- 情報の器を用意する… **データ型・変数・配列**、そして**ポインタ**
- プログラムの流れの制御… さまざまな**アルゴリズム**と**関数**
- データをまとめて扱う **構造体**

技術評論社

【C言語の全貌が理解できる2部構成】

本書では、「プログラミングとは?」という基礎の基礎から出発し、抽象的な座学にとどまらずにC言語のしくみまで踏み込んで学習します。
後半部は、実践演習できるコンピュータ環境で、プログラムを実行しながら読み進むことをお勧めします。万一読解のみでも十分理解に耐えるよう、短いサンプルプログラムを用いて解説しました。

第1部　C言語の基礎知識

- ・プログラミングとは何をするものなのか
- ・C言語の特長
- ・コンパイル環境を準備する

第2部　C言語プログラミングの実際

- ・プログラム例＋実行結果
  …データ型～変数～配列～ポインタ
  アルゴリズムと関数～構造体

Cover Illustration ● Masahiro Yoshida
Cover Design ● KuwaDesign

まるごと図解

最新C言語がわかる



高田美樹◉著

技術評論社

はじめに

最近のIT（Information Technology）の進歩は、本当にすさまじいものがあります。次々と新しい技術が開発され、ハードウェアが改善され、ソフトウェアが作られています。けれども、そういった技術革新の波が、しっかりした基本技術の上に成り立っていることは、いうまでもありません。本書は、ソフトウェア開発の基礎の基礎・プログラミングの考え方について、プログラミングが初めての読者を対象に、やさしく解説したものです。

現在、プログラミングを行うための言語はたくさん開発されていますが、その中でも、C言語は、基本となる言語です。本書では、プログラミングの手段としてC言語を用いていますが、C言語の文法だけをマスターすればよいというのではなく、C言語を通して、プログラミングを行うための環境整備、開発の手順、考え方を学んでいただくことを主眼としております。

優れたIT技術者の育成が叫ばれている昨今、本書が、読者の皆さんにとって、IT技術者への第一歩となりますことを、また、もっと気楽に、プログラミング世界の扉を開くきっかけとなりますことを、願っております。

最後になりましたが、本書を出版する機会を与えてくださり、また、遅々として進まぬ執筆を見守ってくださった技術評論社の方々には、大変お世話になりました。ここに、厚く御礼申し上げます。

2002年8月　高田美樹

contents

contents

5章 プログラムの流れを制御してみよう 147

contents

6章 構造体とファイル処理 241

私たちは、あたりまえのように［スタート］ボタンやアイコンをクリックして、ワープロで文書を作ったり、インターネットを楽しんだりしています。が、その裏では、いったいどんなこ

プログラムはどうして動くのか

とが行われているのでしょうか。プログラミングを始める前に、プログラマがぜひ知っておきたい、コンピュータの動作のしくみについて、ソフトウェアを中心にまとめておきましょう。

1章

1-1 ハードウェアとソフトウェア

パソコンで何か作業をするとき、何が必要なのでしょうか？　「パソコン」と呼ばれる「箱」には、実にさまざまな電子部品が詰まっていますが、それだけでは、何もできません。ハードウェアとソフトウェアは、互いに密接な関係にあるのです。

●ハードウェアとは？

「さあ、これからパソコンを使いましょう。何が必要ですか?」まず、何といってもパソコン本体。これがなければ始まりません。それから、情報を表示するためのモニタ。情報を入力するためのキーボードとマウス。ここまでは不可欠ですね。さらに、印刷したければ、プリンタ。画像を用意したければ、デジタルカメラやスキャナ。電話回線を通してインターネットをしたければ、モデム…。さまざまな機器が必要になってきます。これらの機器を**周辺機器**といいます。

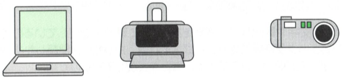
さて、パソコン本体には、何が入っているのでしょうか?　最も重要なものだけを挙げれば、以下のようになります。

・情報を処理するための中央処理装置（CPU）
・情報を処理する際に一時記憶しておくメモリ（RAM）
・情報を記録しておくためのハードディスク
・各装置を結ぶ通路（バス・ポート）

どの装置も、パソコン本体のフタをあければ、これがハードディスク、これがメモリ、と指すことができます。周辺機器も含め、このように目に見える装置を**ハードウェア**といいます。

▼パソコン作業に必要なもの

まずパソコン　　それからプリンタ　　場合によってはデジカメetc…



●ソフトウェアとは？

必要なハードウェアは揃いました。コンセントを差し込み、電力の供給もOKです。でも、まだ足りないものがあります。それは、ソフトウェアです。たとえば、インターネットでネットサーフィンしたければ、ホームページを表示するためのブラウザと呼ばれるソフトウェアが必要です。ワープロで文書を作成したければワープロのソフトが、計算書を作成したければ表計算ソフトが、ゲームで遊びたければゲームソフトが必要です。それだけではありません。マウスがクリックされたとか、キーボードのどのキーが押された、といったことを察知して必要な情報をソフトウェアに通知したり、ハードディスクやフロッピーディスクから情報を読み出したり、といったパソコンそのものを動かすためにもソフトウェアが必要なのです。

では、ソフトウェアって何でしょう？　一言で表すと、

「コンピュータが動作するための手順書の集まり」

です。「手順書」ですから、その実態は目に見えません。目に見えるハードウェアに対し、目に見えないものを**ソフトウェア**といいます。

ソフトウェアのうち、ワープロやブラウザなど、直接作業をするためのソフトウェアを**アプリケーション・ソフトウェア**、パソコンそのものを動かすためのソフトウェアを**基本ソフトウェア**、または、**オペレーティングシステム（OS）**といいます。代表的なパソコン用オペレーティングシステムに、Windows、Linux、MacOSなどがあります。

1-2 コンピュータの頭の中を覗いてみると

コンピュータは、文字や数値などの情報をどうやって理解・記憶しているのでしょうか？　コンピュータが理解できることは、それは、たったの2種類しかありません。'0'と'1'です。この2つだけですべての情報を表現しているのです。

●'0'と'1'だけの世界

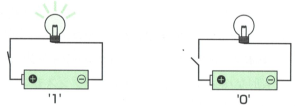
今ここに、豆電球とスイッチ、それに乾電池が1つずつあったとします。これらを図のように直列につないだとき、スイッチを閉じれば豆電球は明るく光り、スイッチを開けば豆電球は消えます。つまり豆電球は点いているか、消えているか、どちらかの状態しかありません。

この状態をそのまま保っていること、これがすなわち情報を記憶しておくことなのです。いま、スイッチONで豆電球が点いている状態を**'1'**、スイッチOFFで豆電球が消えている状態を**'0'**とすれば、0または1を記憶したことになります。もちろん、パソコン内のメモリに乾電池と豆電球とスイッチが入っているわけではありません。実際には、IC（Integrated Circuit：集積回路）と呼ばれる電子部品の出力電圧が0Vか＋5Vか、のどちらかで2つの状態を識別します。

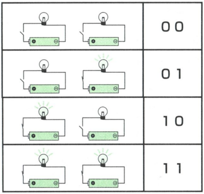
●2進数とは

豆電球が1つあると、**'0'**と**'1'**という2種類の情報を表すことができます。

▼0か1か



▼0と1の組み合わせ



もっと多くの情報を表すにはどうしたらよいのでしょうか？　豆電球を2つ用意してみましょう。上の表のように4つの状態を表すことができます。

豆電球の数を増やせば、それだけ多くの情報を表現することができます。このように、'0'と'1'というたった2種類の数字を並べて表された情報を**デジタル情報**といい、'0'と'1'の数字の列を**2進数**といいます。

さらに、2進数の1ケタを**1ビット**といいます。ビットは、情報の大きさを表す最小の単位です。また、8ビットをまとめて**1バイト**といいます。1バイトを構成する各ビットには、下図のように、順にビット番号がつけられています。

▼1バイトのしくみ

ビット番号→	7	6	5	4	3	2	1	0

2ビットでは、上の豆電球の図のように

$2^2=4$（通り）

の情報を表現することができます。同様に、nビットでは、2^n通りの情報が表現できます。1ビット増すごとに、扱える情報の量は2倍に増えていきます。コンピュータ内部のメモリには、膨大な数の'0'と'1'が記憶されているのです。

●2進数と16進数

もっと大きな数値を2進数で表してみましょう。

0100 0001 1010 0111

このような'0'と'1'の数字の列は、人間にとっては、あまりありがたくありません。まちがいのモトです。そこで、4ビットずつまとめて右の表のように16進数で表します。16進数は、0～15まで16通りを1ケタで表現するものですが、数字は0～9までの10通りしかありません。そこで、11番目をA（またはa)、12番目をB（またはb）…16番目をF（またはf）と書きます。また、16進数であることがわかるように、先頭に0xをつけて書くこともあります。16進数を使うと、上の2進数は

0x41a7

と書けます。少しは、人間にやさしくなりましたね。

▼16進数と2進数

16進数	4桁の2進数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

●2進数で表す文字

ビット数を増やしていけば、もっともっと大きな数値でも、2進数で表せることはわかりました。それでは、'a'や'b'などの文字はどうやって2進数で表すのでしょうか? これについては、国際的にコードが決められていて、7ビットで1文字を表すことになっています。これを**ASCIIコード**といい、128文字が規定されています。実は、1文字は1バイト（8ビット）で表しますので、1ビット余っています。この余った部分は、国ごとに自由に決めてよいことになっています。日本では、カタカナが割り当ててあり、カタカナ部分を含めてJISコードと呼ばれています。ところが、日本語の場合には、ひらがながあります。漢字もあります。とても1バイト（＝8ビット：256通り）では表現しきれません。そこで、2文字分をくっつけて2バイトで表現することになりました。このような文字を2バイト文字といって、いわゆる全角の文字がこれにあたります。

▼ ASCIIコード表

4~7ビット / 0~3ビット	0000	0001	0010	0011	0100	0101	0110	0111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	¥	l	¦
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

これに対して、1バイトで表すことのできる文字が、半角と呼ばれているわけですね。全角のカタカナは2バイト文字ですが、半角のカタカナは、1バイトで表されるJISコードのうち、日本特有の部分というわけです。

One Point 用語解説

➡デジタルとアナログ

デジタルは、英語のdigital（形容詞）をカタカナ表記したものです。digitは、指という意味の名詞ですが、指で数えることから、数字という意味になったと言われています。コンピュータの世界では、0の次は1であり、0.5や0.8は存在しません。このような量をデジタルといいます。

これに対し、アナログは、波形のような連続する量を扱います。ですから、アナログでは、一度波形がくずれてしまうと、元に戻すのは容易ではありません。しかし、デジタルでは、1に近ければ1、0に近ければ0に修正すればいいのです。デジタル情報がノイズに強いのは、このようなデジタルの性質によるのです。

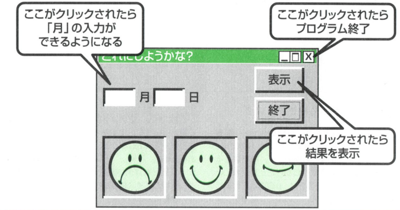
1-3 オペレーティングシステムの役割

ワープロを使って文書を作成しているときも、ブラウザを起動してネットサーフィンを楽しんでいるときも、私たちは、気づかないところで、いつもいつもオペレーティングシステムのお世話になっています。オペレーティングシステムには、大きく分けて3つの役割があります。

●何が命令されたかな？

キーボードのキーを押すと、文字を入力することができますが、たとえば、Cのキーを押したとき、同時に何も押されていなければ小文字の「c」が入力されたことになります。が、Shiftキーが同時に押されていれば、大文字の「C」になります（Caps Lockが解除されているとき）。もし、Ctrlキーが同時に押されていれば、文字の入力ではなく、「選択された情報をクリップボードと呼ばれる特別なメモリに記録する」という意味になる場合もあります。今度は、マウスのクリックを考えてみましょう。下図のように、画面のどこでクリックされたかで、期待される動作は異なります。

▼クリックされた位置と動作

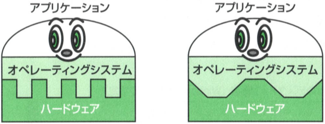


このように、キーの入力、マウスのクリック1つとっても、考えなければならないことがたくさんあります。これらを、それぞれのアプリケーションがいちいちやっていては、とても大変です。オペレーティングシステムは、このような判断を一手に引き受け、結果をアプリケーションに通知します。

●ハードウェアがちがっても

コンピュータの機種がちがえば、ハードウェアはちがってきます。次々に開発される新しい機器に、各アプリケーションがそれぞれ対応しなければならないとしたら…それは膨大な作業量になってしまいます。オペレーティングシステムは、ハードウェアのちがいを吸収し、アプリケーションがどの機種でも同じように動く土台を提供します。

▼オペレーティングシステムの働き



●情報の交通整理

ワープロで文書を作っている最中でも、画面の隅にはデジタル時計が正確に時を刻んでいます。また、プリンタで印刷している間にも、ワープロで次の文章を入力することができます。中央処理装置（CPU）などのハードウェアは1つしかないのに、あたかも同時にいろいろな仕事をこなしているかのようです。また、ハードディスクなどにファイルを記録するときも、ちゃんと空いている領域を探し、他のファイルには一切影響しないように保存できます。このように、情報を整理し、ハードウェアが持っている資源を有効に活用するのも、オペレーティングシステムの仕事です。

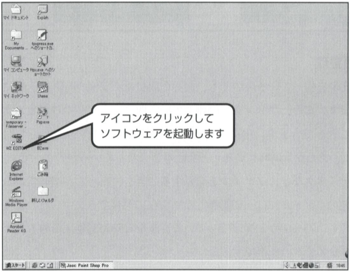
1-4 MS-DOSとWindows

パソコン用のオペレーティングシステムにはいくつかありますが、現在主流となっているのは、アメリカのマイクロソフト社が開発したWindowsシリーズです。そしてその前身は、MS-DOSと呼ばれる、キーボードから命令を打ち込むものです。MS-DOSは、現在でも活躍しています。

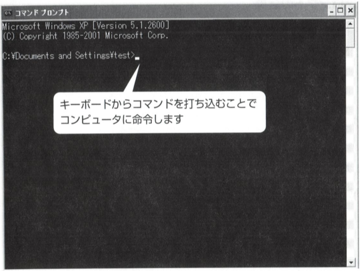
●カラフルなWindows

皆さんもよくご存じのように、Windowsは、主に、画面上に表示されたボタンや、アイコンと呼ばれる小さな絵をマウスでクリックすることで、コンピュータにやりたいことを伝えます。このような操作法を**GUI（グラフィカル・ユーザ・インタフェース）**といいます。私たちユーザは、画面に並べられたメニューやアイコンの中からやりたいことを探してクリックします。けれどもこれは、言い換えると「メニューやアイコンにないことはできない」ということなのです。

▼ Windowsのデスクトップ画面（Windowsクラシック設定）



▼ MS-DOS画面



●MS-DOSは黒い画面

パソコン用のオペレーティングシステムとして、最初に爆発的に普及したのが、MS-DOSでした。MS-DOSは、上図のような真っ黒い画面ですべての作業を行います。そのため、基本的にはマウスは使えず、キーボードから**コマンド**と呼ばれるMS-DOSに対する命令を打ち込みます。どんなことができるのか、そのためにはどうしたらよいのか、をすべて知っていなければMS-DOSを自在に操ることはできません。が、一連の操作を自動化できるなど、熟練すればWindowsにはない良さがあります。このように、私たちユーザが、文字を使ってコンピュータに命令する方法を**CUI（キャラクタ・ユーザ・インタフェース）**といいます。

現在でも、MS-DOS上でなければ動作できないソフトウェアがたくさんあります。そのため、MS-DOSは、「コマンドプロンプト」「MS-DOSプロンプト」としてWindowsの中に組み込まれています。

●コマンドプロンプトの起動

それでは、ためしにWindowsからコマンドプロンプト（MS-DOSプロンプト）を起動して、MS-DOSを体験してみましょう。なぜそんなことをするのか…それは、これからの皆さんのプログラミング学習にMS-DOSが必要だからです。

・Windows98／NTの場合

Windows画面左下の［スタート］ボタンをクリックし、［プログラム］－［MS-DOSプロンプト］をクリック

・WindowsME／2000の場合

Windows画面左下の［スタート］ボタンをクリックし、［プログラム］－［アクセサリ］－［MS-DOSプロンプト］（ME）－［コマンドプロンプト］（2000）をクリック

・WindowsXPの場合

Windows画面左下の［スタート］ボタンをクリックし、［すべてのプログラム］－［アクセサリ］－［コマンドプロンプト］をクリック

さて、起動されたコマンドプロンプトの画面には、

```
C:¥Documents and Settings¥test>
```

または、

```
C:¥WINDOWS>
C:¥
```

などと表示され、カーソルが点滅しています。MS-DOSでは、カーソルが点滅しているときが、ユーザからのコマンドの入力を受け付けるときです。点滅しているカーソルより左側に表示されている＞までの部分を**プロンプト**といいます。

では、何かコマンドを入力してみましょう。キーボードから

```
time [Enter]
```

と打ち込んでみてください。画面には、次の図のように表示されます。

```
C:¥WINDOWS>time
現在の時刻は 5:44:20.5
時刻を入力してください：_
```

プロンプト

timeは、パソコン内部の時計が示す現在の時刻を表示させるコマンドです。もし時刻がまちがっていたら、正しい時刻を入力してください。まちがっていなかったら、そのまま［Enter］を押します。

```
C:¥WINDOWS>time
現在の時刻は 5:44:20.5
時刻を入力してください：5:49:00 [Enter]
C:¥WINDOWS>_
```

正しい時刻を入力された方は、Windows画面右下の時刻表示が正しくなったことに気づかれたことでしょう。

次に、

exit［Enter］

と入力してみてください。コマンドプロンプトは終了し、黒いウィンドウは消滅します。

このように、MS-DOSでは、キーボードからさまざまなコマンドを入力することで、操作するのです。

1-5 ソフトウェアの正体

ソフトウェアは、「コンピュータが動作するための手順の集まり」であるということは、すでにお話ししました。コンピュータは、'0'と'1'だけで表現された膨大な手順書の集まりを1つ1つ解釈して実行していくのです。

●2進数で表された手順

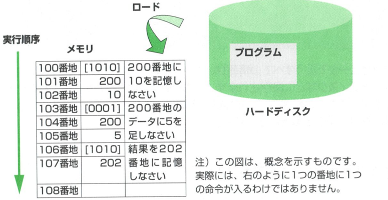
コンピュータは、'0'と'1'しか理解することができません。従って、動作の手順、つまりソフトウェアも、'0'と'1'だけで表現しなければなりません。文字を2進数で表現するためのコード表があるように（P.17参照）、手順もこれを2進数で表現するための変換表があります。ただし、文字コードがASCIIやJISで規格として統一されているのに対し、手順の変換表は、ハードウェア、もっと厳密にいえば、CPUごとに異なっているのです！　なぜなら、変換した結果、動作を行うのがCPUであり、変換表は、CPUの構成（ハードウェア）と直結しているからです。

▼架空CPUの手順変換表

コード	動作
0001	足し算しなさい
0010	引き算しなさい
0011	掛け算しなさい
0100	割り算しなさい
0101	1ビット右シフトしなさい
0110	1ビット左シフトしなさい
1001	データを読み込みなさい
1010	データを記憶しなさい

いろいろな目的のために、「手順」をら列したものを**プログラム**といいます。

▼ プログラムのロード



100番地	[1010]	200番地に
101番地	200	10を記憶し
102番地	10	なさい
103番地	[0001]	200番地の
104番地	200	データに5を
105番地	5	足しなさい
106番地	[1010]	結果を202
107番地	202	番地に記憶しなさい
108番地		

注）この図は、概念を示すものです。実際には、右のように1つの番地に1つの命令が入るわけではありません。

●プログラムの実行

プログラムは、通常はハードディスクなどに記憶されています。そして、必要になったとき、たとえば、Windows上のアイコンがダブルクリックされて、ソフトウェアが実行されることになったとき、メモリに読み込まれます。メモリに読み込む作業を**ロード**といいます（上図）。

メモリにロードされたプログラムは、番地の小さい方から順に1つずつCPU内で解釈されます。上の例では、表の[ ]の部分が、手順変換表により変換されて、命令が実行されていくのです。実際には、もっと多くの手順を踏んで、キーボードから入力したり、画面に表示したりといった複雑な操作ができます。

ソフトウェアを終了すると、プログラムに占領されていたメモリは開放され、また、新たに実行される別のプログラムが利用できるようになります。ソフトウェアによっては、常にメモリ内にロードされていて、いつでもすぐに実行できる状態になっているものもあります。そのようなソフトウェアを**常駐プログラム**といいます。常駐プログラムが多いと、メモリが占有され、他のプログラムをロードできる領域が少なくなってしまいます。

1-6 ファイルとフォルダ

コンピュータで扱うすべての情報は、ファイルという単位で、ハードディスクやフロッピーディスクなどに保存されます。そのため、プログラムもデータも見た目は皆同じです。これらの膨大なファイルを広大なディスクに整然と保存するため、フォルダと呼ばれる小部屋を用意します。

●あれもファイル・これもファイル

ワープロを使って作成した文書は、ファイルとしてディスクに保存します。文書ばかりではありません。デジカメで撮った写真やお絵描きソフトで描いた絵もファイルとして保存しますし、デジタル化された音楽もファイルとして保存します。ファイルとは、これらの情報を保存するための最小単位なのです。書類を項目別にファイリングするのとよく似ていますね。

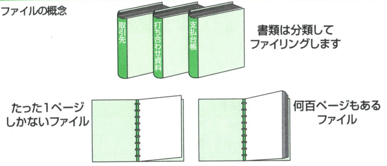
書類ファイルには、たった1枚しか書類がないファイルもあれば、何百ページにもわたるファイルもあります。コンピュータで扱うファイルも同じです。たった1つの画像も1つのファイルですし、何百ページにもわたるワープロ文書も1つのファイルです。

●こんなのもファイル

今までの話題は、すべて「データ」を保存することを目的とした「ファイル」でしたが、実は、ワープロやインターネット、ゲームなどをするためのソフトウェア自身も「ファイル」という形態でハードディスクなどに保存されています。そして、必要なとき、つまり、ソフトウェアが起動されたとき、メモリに読み込まれて実行を開始することは、前節で学んだとおりです。

このように、ファイルは、「データファイル」と「アプリケーションファイル」に大別されます。アプリケーションファイルのように、実行できるファイルを**実行可能ファイル**といいます。

▼ファイルの概念



●ファイルを区別する拡張子

1台のマシンには、実に多くのファイルが保存されています。これらは、**ファイル名**と呼ばれる、ファイル1つ1つにつけられた名前で区別されます。これほどの多彩なファイルですから、このファイルがどんなファイルなのか、データファイルなのか、アプリケーションファイルなのか、また、同じデータファイルでも、画像なのか文書なのか、文書であれは、どのワープロソフトで作られた文書ファイルなのか、ファイル名を見ただけで識別できれば便利です。

そこで、ファイル名は、下図のように、ピリオドで区切られた2つの部分から構成されています。前の部分はファイル固有の内容を表す名前で、後ろの部分はこのファイルの性質を表す3文字の記号です。後者を**拡張子**といいます。拡張子によって、このファイルがどんなファイルなのかを見分けることができます。

▼ファイル名の構成



下表は、拡張子の一例です。データファイルを生成するアプリケーションソフトは独自の拡張子を定めていることが多く、拡張子の種類は、実にたくさんあります。

▼拡張子の一例

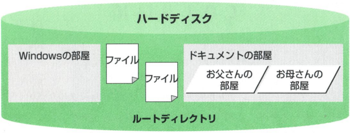
拡張子	意味	拡張子	意味
exe	実行可能ファイル	txt	テキストファイル
bat	バッチファイル	bmp	画像ファイル
bak	1つ前のファイル	jpg	画像ファイル
zip	圧縮ファイル	gif	画像ファイル
lzh	圧縮ファイル	ico	アイコン
c	C言語ソースファイル	htm	HTMLファイル
cpp	C++ソースファイル	cgi	web上で動くプログラム
inf	設定情報	hlp	ヘルプファイル
ini	設定情報	tmp	一時ファイル

アプリケーションをマシンにインストールしたとき、そのアプリケーションで扱うファイルの拡張子を登録しておくことで、データファイルとアプリケーションとを関連付けることができます。このようにしておくと、データファイルをダブルクリックするだけで、必要なアプリケーションを起動させたり、データファイルのアイコンとして特定のものを表示したりできるようになります。

●フォルダ（ディレクトリ）

大容量のハードディスクにファイルを保存するとき、無秩序に保存していたのでは、何をどこに記録したのかわからなくなってしまって、探すのが大変です。ちょうど、東京ドームに段ボール箱をどんどん投げ入れるようなものです。そこで、広い空間を小さな部屋に区切って、目的に応じて分けて使うことが考えられました。たとえば、Windowsが使う部屋、私たちユーザが作った文書や画像などのドキュメントをしまっておく部屋…。この小さな部屋を**フォルダ（ディレクトリ）**といいます。また、部屋の中をさらに小さな部屋に分けることもできます。ドキュメント部屋の中に、お父さんが使う部屋、お母さんが使

▼階層構造



う部屋、子供が使う部屋を設ける、といったぐあいです。お父さんの部屋には、仕事の部屋、趣味の部屋、個人資産を管理する部屋、などがあるかもしれません。このようなフォルダの構造を**階層構造**といい、一番外側のフォルダを**ルートディレクトリ**といいます。

One Point 用語解説

➡絶対パスと相対パス

ファイルやフォルダを指定するとき、同時にその所在を表す方法をパスといいます。たとえば、上の図で、お父さんの部屋に「日記.doc」というファイルが保存されていたとします。これは、お父さんがつけている日記です。お母さんの部屋にも、「日記.doc」というファイルがあったとします。これは、お母さんがつけている日記です。両者のファイル名は同じですが、中身は異なります。このように、ファイル名が同じであっても、保存されているフォルダが異なれば、まったく別のファイルとして扱わなければなりません。パス名を使うと、名前が同じこの2つのファイルを区別することができます。

今、ハードディスクがCドライブに割り当てられているとき、

c:¥ドキュメントの部屋¥お父さんの部屋¥日記.doc

のように、ルートディレクトリからたどったパス名を絶対パスといいます。これに対して、現在位置（カレントディレクトリ）からの相対的な位置を表すパス名を相対パスといいます。このとき、自分より外側のフォルダは..で表します。たとえば、「お母さんの部屋」を起点として、お父さんの日記ファイルを相対パスで表すと、

..¥お父さんの部屋¥日記.doc

となります。

COLUMN

MS-DOSのコマンド

MS-DOSでは、プロンプトが表示され、その脇でカーソルが点滅しているときに、キーボードからコマンドを入力してコンピュータに仕事を命令します。コマンドは、内部コマンドと外部コマンドに分けられます。

内部コマンドは、COMMAND.COMというファイルに内蔵されたコマンドで、MS-DOSが起動されたときにメモリに読み込まれ、常駐します。ですから、内部コマンドはMS-DOS上でいつでも使うことができます。

▼主な内部コマンド

コマンド	意味
dir	プロンプトと一緒に表示されているディレクトリ（カレントディレクトリといいます）に含まれるファイルとディレクトリの一覧を表示
cd　ディレクトリ名	カレントディレクトリを指定のディレクトリに変更
del　ファイル名	指定のファイルを削除
date	コンピュータが保持している現在の日付を表示。変更も可
time	コンピュータが保持している現在の時刻を表示。変更も可

外部コマンドは、MS-DOS上で実行できる実行可能ファイルを実行するものです。ですから、MS-DOSが探すことのできるフォルダに

　　　コマンド名.exe

という名前のファイルが存在しなければなりません。

COLUMN

第１章では、コンピュータの内部には、'0'と'1'が詰まっていることを学びました。プログラムもデータも'0'と'1'のかたまりでした。
まさか、プログラミングって、

プログラミングとは何をするものなのか

'0'と'1'を並べる作業ですか?そう、昔のプログラマは、そういう作業をやっていました。が、それは、人間にとっては、とても大変な仕事です。ここでは、人間が、'0'と'1'から開放されるしくみを学びましょう。

2章

2-1 あそこにも、ここにもプログラム

現在、私たちは、毎日コンピュータのお世話になって生活しています。そして、世の中には、実にさまざまな形態のソフトウェアがあります。プログラミングの学習の始めに、ソフトウェアがどんなところで活躍しているのかを知っておきましょう。

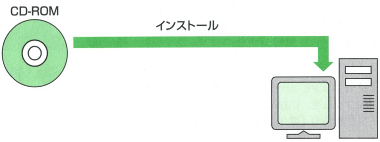
●パソコンで

パソコン上で動くプログラムは、おなじみですね。基本ソフトウェア（オペレーティングシステム）と、アプリケーション・ソフトウェアとに分けられることは、すでに学びました。パソコン用のソフトウェアの多くは、CD-ROMやフロッピーディスクに記録されて供給されます。ソフトウェアをパソコンで使うためには、あらかじめ、CD-ROMやフロッピーディスクからパソコンにインストール、つまり、供給された情報をパソコンのハードディスクにコピーし、登録しなければなりません。この作業は、初めに一度だけしておけば、あとはいつでもプログラムを使うことができますし、多くの場合、CD-ROMやフロッピーディスクは、もう必要ありません。

●家庭用ゲーム機で

家庭用ゲーム機では、ゲームソフトはカセットに収められているタイプもあります。このようなタイプのゲーム機では、カセットを機械にセットしさえすれば、ゲームを楽しむことができます。あらかじめインストールする必要はありませんが、ゲームをするときには、必ずカセットが必要になります。また、このカセットは、異なるタイプの機械では使うことができません。このように、特定の機械専用のソフトウェアもあります。

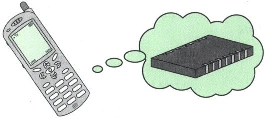
▼CD-ROMからパソコン本体へ、ソフトウェアのインストール



●携帯電話や炊飯器で

最近の携帯電話には、さまざまな機能が付いています。時計や電卓の役目もできますし、メールをやりとりしたり、ホームページを見たりすることができる機種もあります。これらのプログラムは、CD-ROMやフロッピーディスクからインストールする必要はなく、買ったその瞬間から使うことができます。炊飯器も同様です。タイマをセットして、好みの時間に炊き上げるのは、実は、プログラムのなせる技なのです。これは、**ROM**（Read Only Memory）と呼ばれる、半永久的に情報を保存できる記憶素子にプログラムが焼き付けられていて、いつでも、実行できるようになっているからです。このような、特定の機械にはじめからセットされているプログラムは組み込みプログラムといって、これも立派なソフトウェアです。ROMに対し、パソコンのメモリに使われている、電源を切るとデータが失われてしまう記憶素子を**RAM**（Random Access Memory）といいます。

▼家電にもプログラム



ROMにプログラムが記録されています

2-2 いろいろなプログラム言語

コンピュータに対する命令は、コンピュータが理解できる「言葉」で伝えなければなりません。コンピュータが理解できる言葉をプログラミング言語といいます。複雑なプログラムを、いかに簡単に、いかに速く、いかに確実に書くか、それがプログラミング言語の「進化」の原動力なのです。

●機械語

コンピュータが理解できるのは、'0'と'1'だけであって、数値も文字もコンピュータに対する命令も、あらゆる事柄は、'0'と'1'のら列で表現されなければならないことは、すでに学びました。コンピュータが理解し、実行できる'0'と'1'のら列で書かれたプログラム言語を**機械語**といいます。昔の偉大なプログラマたちは、'0'と'1'をコンピュータに入力することによって、プログラムを作っていたのです。

●アセンブリ言語

機械が理解できるのは機械語ですから、プログラムを機械語で書くことは、機械にとってはありがたいことでしょうけれども、人間にとってはとても辛い作業です。'0'と'1'を1箇所まちがえただけでも、コンピュータは思うとおりには働いてくれないのですから！　そこで、命令を人間がわかりやすい記号に置き換えることで、人に優しい工夫がなされました。たとえば、足し算をさせた

▼機械語のプログラム

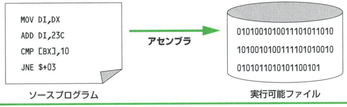
```
0100 0011 1010 0101 1111 1000
0100 0000 0000 1101 1100 1111
1010 0110 0101 1110 1110 1111
```

いときは、「0001」と書くかわりに「ADD」と、引き算をさせたいときには、「0010」と書くかわりに「SUB」と書ければ、少しは楽です。もっとも私たち日本人には、これでもまだ不満ですが、コンピュータが英語圏で発達したものである以上、これはしかたのないことでしょう。このような、人間にわかりやすい記号を使って書くプログラム言語を**アセンブリ言語**といいます。アセンブリ言語で書かれたプログラムは、**ソースプログラム**といって、テキストファイルとして保存します。ですから、Windowsの付属ツールである［メモ帳］などのテキストエディタを使って、編集することができます。

けれども、アセンブリ言語のままでは、コンピュータは命令を理解できません。コンピュータはあくまでも、'0'と'1'のら列、機械語でないと理解できないのですから。それに、テキストファイルは、すべてアスキーコードで記録されており、数字の'1'は、「0000 0001」ではなく、「0011 0001」となっています。数字を、数値に直さなければコンピュータは処理できないのです。

そこで、アセンブリ言語で書かれたプログラムを機械語に翻訳するプログラムが開発されました。これをアセンブラといいます。アセンブラは、テキストファイルを読み取って、実行可能ファイルを生成します。実行可能ファイルは、テキストではありませんので、内容をテキストエディタで確認することはできません。が、これをメモリにロードすることで、コンピュータに実際に仕事をさせることができるのです。

▼アセンブラで機械語に翻訳



アセンブラが開発されたことによって、人間は、'0'と'1'のら列から開放されることになりました。

●高級言語

アセンブラは、機械語を単に記号に置き換えるだけですから、機械語が異なれば、アセンブリ言語も異なります。また、日常私たちが使っている
「もし、平均点が60点より大きかったら、評価は合格とする
それでなければ、不合格とする」
などという“言葉”とはかけはなれているものですし、ちょっとした数式、

$$y = x1 + x2 + x3$$

なども、そのまま書くことはできません。もっともっと人間の言葉に近い言語でプログラムを書きたい。もっともっと効率を上げたい。もっともっと楽をしたい!… そんな人間の欲求と、どんどん複雑なプログラムを書かなければならなくなった必然性から開発されたのが、高級言語です。コンピュータの発達とともに、コンピュータにさせたい仕事の内容も多様化してきたため、それぞれの仕事に向いた多くの高級言語が開発されました。アセンブリ言語と同様、高級言語で書かれたプログラムはソースプログラムと呼ばれ、テキストファイルとして保存されます。これを機械語に翻訳し、実行可能ファイルを生成するソフトウェアを**コンパイラ**といいます。

▼コンパイラで実行可能ファイルに変換



●多種多様な高級言語

プログラムの開発が多岐にわたるにつれ、その目的ごとに、多くの高級言語が開発されました。その一部を紹介します。

▼主な高級言語

BASIC	Beginner's All-purpose Symbolic Instruction Code／初心者向き多目的な言語。
COBOL	COmmon Business Oriented Language／給与計算などの事務処理用に開発された言語。大型計算機全盛時代には、花形だった。
FORTRAN	FORmula TRANslater／科学技術計算用に開発された言語。数式がほぼそのまま書ける。
Pascal	Philips' Automatical Sequence Calcurator／教育用言語。
ALGOL	ALGOrythmic Language／アルゴリズムの記述に向いている。PsacalやC言語に影響を与えたといわれている。
LISP	LISt Processor／人工知能研究などで利用されることが多い。
Prolog	PROgramming in LOGic／人工知能の開発に適している。
C言語	オペレーティングシステム用に開発された言語だが、その利便性から、あらゆるプログラムの開発に利用されている。Windows以前のMS-DOSの時代には、一世を風靡した。
C++	C言語にオブジェクト指向的な考え方を追加した言語。オペレーティングシステムからアプリケーションまで、また、大規模なプログラムをも開発できる。
Java	C++に似ている。オブジェクト指向の考え方を持つ上、セキュリティを強化するなど、ネットワーク環境を強く意識している。

One Point 用語解説

➡テキストとバイナリ

文字だけで構成されたファイルをテキストファイルといいます。「文字」とはいえ、その正体は、アスキーコードで表された'0'と'1'の羅列です。テキストエディタは、1バイト（8ビット）を束にして、アスキーコード表で変換した文字を表示するのです。

テキストファイルは、文字のみ、すなわち、アスキーコードだけを含んだファイルですから、どんな機種のコンピュータでも扱うことが可能です。これに対し、特定の約束をした、アスキーコード以外の情報を含むファイルをバイナリファイルといいます。バイナリファイルの表示には、ファイルの約束を理解できる専用のソフトウェアが必要です。

2-3 コンパイラ言語とインタプリタ言語

プログラムを実行するには、コンパイル、メモリにロード、そして実行、という手順を踏みます。この過程で、全部まとめて翻訳し、まとめてロード、一気に実行するという基本的な手順のほかに、必要になった部分だけを、そのつど翻訳し、実行するという方法もあります。

●コンパイラ言語

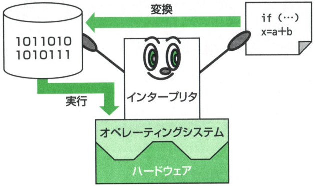
コンパイラ言語は、あらかじめソースファイルを丸ごと全部翻訳し、完全な実行可能ファイルを生成します。そして、それを全部（コンピュータのメモリがある限り）ロードし、一気に実行します。ですから、万が一、その言語で定められている規則（文法）どおりにソースプログラムが書かれていないなどの理由により、コンパイラが翻訳できなかった場合には、実行可能ファイルが生成されません。たとえ何万行もあるような大きなプログラムのうちの、たった1行でもコンパイラが解釈できない文があるとダメなのです。もし、その誤りがある部分が、絶対に実行されない個所（プログラムの組み方とデータによっては、実行されない文があることがあります）だったとしても、ダメなのです。プログラムを実行してみるためには、全ソースプログラムが完璧な文法で記述されていることが必要不可欠です。そのかわり、実行可能ファイルさえ生成されて実行が始まってしまえば、余計な作業をしませんから、非常に高速に処理をすることができます。

▼ コンパイラ言語をたとえると…



コンパイラ言語は、高速なジェット機。飛び立ってしまえば
高速に移動できるが、飛び立つまでに時間がかかる。

▼インタプリタの働き



●インタプリタ言語

コンパイラ言語に対し、実行する必要が生じたときに、その部分だけを翻訳してロードし実行する方式の言語を**インタプリタ言語**といいます。あらかじめすべてを翻訳するということはしませんので、ソースプログラムを書いたら、その場でためしに実行してみることができます。ソースプログラムが不完全であっても、途中までしか書いてなくても、とにかくそこまでは実行できるのです。では、どうしてそんなことができるのでしょう? 機械語でなければコンピュータは理解できないはずなのに! それは、オペレーティングシステムとソースプログラムとの間に、別のソフトウェアを介在させることで可能となります。つまり、コンピュータにとって、実行しているのは、ターゲットとなるソースプログラムではなく、間を取り持つ、機械語に翻訳されたソフトウェアなのです。このソフトウェアが、ソースプログラムを1行読み取って翻訳し、間接的にコンピュータに実行させているわけです。

このように、インタプリタは、翻訳しながら実行していきますので、処理速度は遅くなります。しかし、開発の過程において、プログラムの動作を実際に確認しながらプログラムを作っていくことができる、という大きな利点があります。最後に完成したら、コンパイルしておけばいいのです。現在、代表的なインタプリタ言語にVisual Basicがあります。

2-4 イベントドリブン型と手続き型

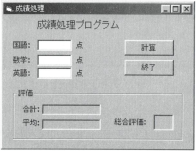
Windowsでは、ユーザのマウス操作により、何がどんな順で入力されたり、ボタンがクリックされたりするかわかりません。そのようなユーザの操作に対応したプログラムをイベントドリブン型といいます。これに対し、MS-DOSでは、プログラム主導で実行が進んでいきます。これを手続き型といいます。

●イベントドリブン型のプログラム

たとえば、「国語・数学・英語の点数を入力して、合計点・平均点・評価を求める」プログラムを考えてみましょう。Windows上で動作する場合、下図のような画面になるでしょうか。

このプログラムは、「国語・数学・英語」の得点の入力をユーザ（パソコンを操作している人）の好きな順で入力することができます。最初に数学の得点を入力することもできます。英語が最初でもかまいません。そして、計算ボタンがクリックされたら、合計・平均・評価を算出して表示します。このような動作は、プログラムの立場から言い換えると、入力がどういう順でなされるかわからないということです。もしかしたら、得点の入力を終了する前に「計算」ボタンがクリックされてしまうかもしれません。このような、ユーザ主導で実行するプログラムを**イベントドリブン型プログラム**といいます。

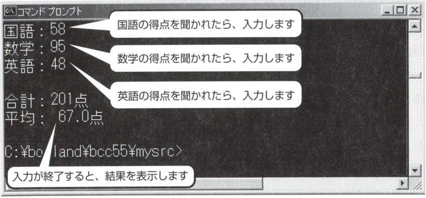
▼ 成績処理プログラムは、たとえばこんな画面に…



●手続き型のプログラム

MS-DOS上で動くプログラムの多くは、マウスを使うことができず、決められた順に入力したり、結果を表示したりします。

▼手続き型プログラムの実行



上のプログラムでは、初めに、何がなんでも国語の得点を入力しなければなりません。ユーザが「数学の得点を先に入力したい」と言っても、ダメなのです。プログラムが、「国語の得点は?」と聞いているのですから、国語の入力です。このように、プログラム主導で実行が進んでいくプログラムを**手続き型のプログラム**といいます。

これから、皆さんにC言語で作っていただくプログラムは、MS-DOS上で実行する、手続き型のプログラムです。

2-5 設計とは、何を考えればいいのか

プログラムを作る（ちょっとカッコよく、「開発する」）には、利用するプログラミング言語の文法を知らなければなりません。しかし、それだけではプログラムは開発できません。ユーザの要求や予想されるデータなど、プログラムを取り巻く状況を最大限考えること、これが設計です。

●ユーザインタフェース

いま、ある日の過去10年間のお天気の傾向を表示するプログラムをつくることになったとしましょう。下図のように、日付を入力すると、過去10年間で、最も多かった天候を表示します。

▼お天気表示プログラム①



仮に、過去10年間の3月18日のお天気が右の表のようであったとしたとき、「晴れ」4日、「曇り」3日、「雨」3日であることから、最も日数の多い「晴れ」が表示されたものとしています。

年	天気
2000年	晴れ
1999年	曇り
1998年	曇り
1997年	雨
1996年	晴れ
1995年	晴れ
1994年	晴れ
1993年	雨
1992年	曇り
1991年	雨

▶過去10年の3月18日の天気

さて、今、「何月?」と聞かれて「3」と入力していますが、人間のやることには、必ずまちがいがあります。もし、まちがえて「14」と入力してしまったらどうでしょう? 「何日?」と聞かれて「18」と入力していますが、まちがえて「35」と入力してしまったらどうでしょう? 14月35日なんて、ありえません。また、「31日」と入力された場合、「3月」であれば、正しい日付ですが、「4月31日」はありません。さらには、「2月29日」はうるう年ですから、4年に1回しか、お天気の統計を取れないことになります。この日の取り扱いは、いったいどうしたらいいのでしょうか?

このように、ユーザから与えられたデータは、それが妥当なものであるかどうかを必ずチェックしなければなりません。そして、万が一、まちがった入力がなされたら、次はどんな動作をしたらよいかも考えておかなければなりません。たとえば、誤った日付が入力されたら、「そんな日付はありません!」とばかりに処理を停止してしまうのか、それとも「誰にでも誤りはあることだから、もう1度入力し直してね」と正しい日付を入力してもらえるまで繰り返すのか…?

プログラムの「開発」とは、起こり得るあらゆる状況を想定して、あらかじめ、その対策を立てておくことが求められるのです。プログラムのうち、ユーザの入力した情報を処理する部分を、**ユーザインタフェース**といいます。ユーザインタフェースの良し悪しは、プログラムが使いやすいかどうかということと、直接関わってきます。

▼妥当でないデータには、どう対処するか?



●データベース

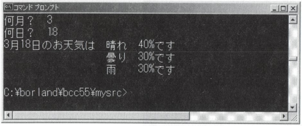
さて、このプログラムでは、3月18日は「晴れ」と表示しました。ということは、3月18日は「晴れ」というデータをプログラム内部で蓄えていなければならない、ということです。それは、3月18日のみならず、入力された日付に応じてその日の天気を表示するわけですから、365日分のお天気のデータを持っていなければならない、ということです。このように、データを蓄えておき、必要に応じて取り出す技術をデータベースといいます。

▼過去10年間のお天気データベース①

1月1日	晴れ				
1月2日	曇り	3月18日	晴れ		
1月3日	曇り	3月19日	晴れ		
1月4日	雨	3月20日	曇り		
1月5日	雨				
:	:		:	:	
				12月31日	晴れ

でも、考えてみてください。3月18日は、10年間のうち、晴れの日は4日、曇りの日が3日、雨の日が3日ですから、「晴れ」が特に多いわけではありません。それでも「晴れ」と表示されてしまうのでは、情報としてちょっと不満な場合も、きっとあるでしょう。そういうときのために、10年間のお天気の割合を表示したらどうでしょう?

▼ お天気表示プログラム②

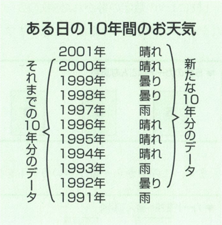


このように表示するためには、左ページの表のデータだけでは足りません。1月1日から12月31日までの、晴れ・曇り・雨の日数の内訳が必要になります。

▼過去10年間のお天気データベース②

日付	晴れ	曇り	雨	日付	晴れ	曇り	雨
1月1日	8日	1日	1日				
1月2日	7日	3日	0日	3月18日	4日	3日	3日
1月3日	6日	2日	2日	：	：	：	：
：	：	：	：				
				12月31日	5日	3日	2日

もし右表のように、翌年になって新しい情報を追加し、10年前のデータを削除したいということになれば、さらに、日数だけではなく、何年の何月何日のお天気が何だったかというデータまで保持していなければならなくなります。



このように、どんな情報を表示するのかを検討し、次に、そのためには内部でどんなデータを持たなければならないのかを考えます。表示したい内容によっては、データの種類や量が異なってくるのです。さらに、それによって、本格的なデータベースソフトを導入しなければならないのか、それとも、テキストファイル程度でよいのかなど、データを保持する手段も決めなければなりません。プログラム開発では、実際にプログラムを書き始める前に、操作性・データベース・拡張性など、さまざまな検討が必要です。設計とは、これらをすべて含むとても重要な作業なのです。

2-6 アルゴリズムとは何か

コンピュータに何か仕事をさせたいとき、どんな手順で、どんな処理を施せば結果が得られるのか、さらに、同じ結果を得るにしても、いかに効率よく得るか、という手順をアルゴリズムといいます。プログラミングを行う上で、アルゴリズムの習得は、非常に重要なことです。

●並べ替え

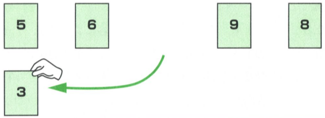
1枚に1つの数字の書かれた5枚のカードがランダムに並べられています。これを小さい順に並べ替えたいと思います。あなたなら、どうしますか?

▼5枚のカードはこんな具合



まず、この中から一番小さいカードを探して先頭に置きます。

▼カードを並べ替えていく



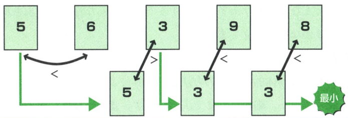
残ったカードの中から、一番小さいカードを探して2番目の場所に置きます。以後、同様に、残ったカードの中から次々と小さいカードを探して、3番目・4番目に置きます。最後に残ったカードは一番大きいカードですから、一番右に置いて、でき上がりです。

一見、完璧な手順のように思えますが、それでは質問です。「一番小さいカード」は、どうやって選びましたか? カードが5枚くらいなら、人間は全体を見渡して、一瞬にして小さいカードを選ぶことができます。では、カードが1万枚あったら? どうやら、一番小さいカードを選ぶ手順は、思ったほど簡単ではなさそうです。

●一番小さいカードを選ぶ

5枚のカードの中から、一番小さいカードを選ぶ手順を考えてみましょう。

▼最小のカードを選ぶ



「5」と「6」とでは、「5」の方が小さい。そこで、今度は「5」と「3」とを比べます。「3」の方が小さいので、次は「3」と「9」とを比べ、小さい方は「3」。最後に「3」と「8」とでは、「3」が小さいので、一番小さいカードは「3」だということがわかります。

●アルゴリズムとアルゴリズムの組み合わせ

一番小さいカードを選ぶ例では、どれとどれとを比べるのかがはっきりしています。それに対して、前述の並べ替える例では、手順がいまひとつ具体性に欠けます。アルゴリズムは、処理の対象を明確にしたものでなければなりません。が、並べ替えの例でも、「一番小さいカードを選ぶ」アルゴリズムができれば、それを組み入れることで具体的な処理になります。このように、処理を部品化し、それを組み立てた**手順**がアルゴリズムなのです。

2-7 プログラム開発の手順

「こんなソフトウェアが欲しいなあ」という漠然とした考えだけでいきなりソースプログラムを書き始めても、よほど小さな趣味的なプログラムでない限り、おそらくうまくはいかないでしょう。設計から開発、そして評価という過程を踏むことで、効率的で無駄のない開発を行うことができます。

開発の手順

①基本計画

まず、どんなソフトウェアを開発するのか、どんな性能が要求されるのかなど、大まかな計画を立てます。

②外部設計

次に、もっと具体的な操作の方法と、それによってどんな結果を得てどのように表示するか、細かい点まで決定します。また、操作に誤りがあった場合を想定した対処法も十分に検討します。

③内部設計

外部設計を実現するには、プログラムをどのような構成にしたらよいのか、どの言語を使うのが最も適しているのか、開発環境は？　内部でどのようなデータを持たなければならないのか、など、ユーザからは見えない内部を考えます。

④コーディング

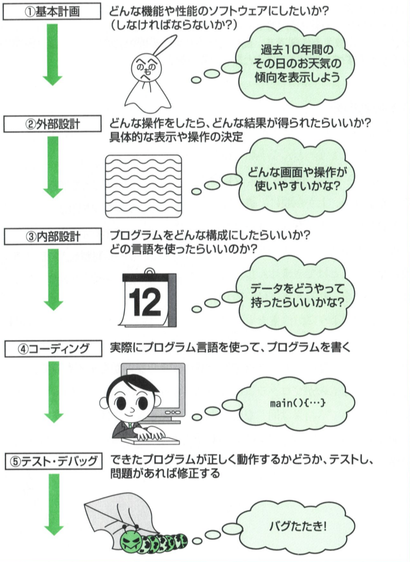
いよいよプログラムを書きます。プログラミング言語の文法にのっとり、アルゴリズムをプログラムで表現していきます。コンパイル言語を選んだ場合には、コンパイルが正常に終了し、実行可能ファイルを生成します。

⑤テスト・デバッグ

できたプログラムを実際に動かしてテストします。期待どおりの結果が得られたでしょうか？　いろいろなデータを使ってテストを繰り返します。期待

に反した結果であった場合には、④、場合によっては③に戻ってやり直します。

▼開発手順、計画からデバッグまで



2-8 プログラムの3つの基本構造

人間は、一瞬の間に、たくさんのことを考え、総合的に判断して行動します。プログラミングは、人間なら何気なくあたりまえのように行っている動作を、1つ1つコンピュータに教える作業です。動作を行うための処理の流れは、たった3種類の組み合わせですべてを記述できると言われています。

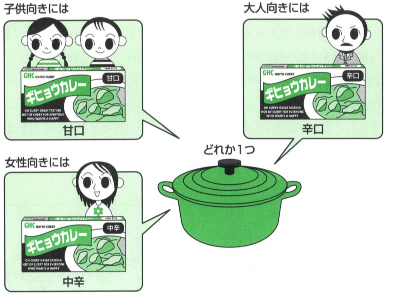
●言われた順に

カレー作り手順書

1. 肉を切る
2. ジャガイモを切る
3. ニンジンを切る
4. タマネギを切る
5. ナベに材料を入れて煮る
6. カレールーを入れて煮る

たとえば、「カレーを作る」作業を考えてみましょう。どんな手順で作りますか？　ざっと、右の表のようになるでしょうか。上から順に、書いてあるとおりに作業をすれば、きっとカレーはできあがるでしょう。材料を切る前に煮込むことはできませんし、先にカレールーを入れると焦げ付いてしまいます。このように、作業をする順序はとても大事で、1つ1つの作業は正しくても、作業の順序をまちがえてしまうと、おいしいカレーはできません。手順書は、どんな作業をどんな順序でしなければならないかを示すもので、この手順書にあたるものがプログラムなのです。もっとも、肉を先に切ってもタマネギを先に切っても結果は同じなのですが、コンピュータは、手順書に書いてある順序を忠実に守って、このとおりに実行します。このように、上から下へ、順序どおりに作業を進めるプログラム構造を**順次構造**といいます。

▼カレーの辛さの選択構造



●どれか1つだけ

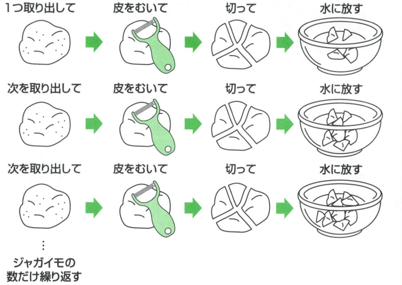
ところで、カレールーには、甘口・中辛・辛口があります。子供向きのカレーを作るなら甘口、甘いものの好きな女性向きなら中辛、刺激を求める大人向きなら辛口がいいでしょう。状況に応じてどのルーを使うか、選ばなければなりません。このように、いつもいつも同じなのではなく、そのときの条件によって、いくつかある中から1つを選ぶプログラム構造を**選択構造**といいます。

選択構造の基本は二者択一です。条件を満たしていれば一方を選び、満たしていないときは他方を選びます。が、条件を組み合わせることで多くの中から1つ選ぶこともできますし、逆に、条件を満たしているときだけ処理を行い、満たしていないときには何もしない、という選択もできます。

●同じことを何度も

ジャガイモを5つ使うとき、「ジャガイモを切る」という動作を5回繰り返します。もっと細かく考えると、「皮をむいて、サイの目に切って、水に放す」という動作を5回繰り返すのです。このように、同じことを繰り返すプログラム構造を**反復構造**といいます。このとき、1つのジャガイモの処理が終わったら、次のジャガイモを手にとって、同じ動作を繰り返しますね。そんなのはあたりまえ、と思われるかもしれませんが、「次のジャガイモ」ということは、案外大事なことなのです。これについては、具体的にプログラムを書くときに考えることにしましょう。

▼ジャガイモ切りの反復構造



●グループにまとめる

カレー作り手順書は、以下のように作業のまとまりを考えてグループを作ることができます。

カレー作り手順書

1. 材料を切る	① 肉を切る ② ジャガイモを5つ切る ③ ニンジンを1本切る ④ タマネギを2個切る
2. 材料を入れて煮る	肉の色が変わり、ジャガイモと ニンジンが柔らかく煮えるまで
3. カレールーを入れて煮る	子供向きなら　甘口 女性向きなら　中辛 大人向きなら　辛口 材料に味が染み込むまで煮る

グループ化すると、全体の処理の手順を大まかにつかむことができます。細かいことはそれぞれのグループの枠ごとに考えればよいのです。C言語では、処理のまとまりを**関数**といいます。

プログラムを設計するとき、まず全体を考え、それから順に詳細を詰めていく設計のしかたを**トップダウン**といいます。逆に、細かいことを思いつくままに書き留めておき、それを体系立てていきながら全体を組み上げていく設計のしかたを**ボトムアップ**といいます。

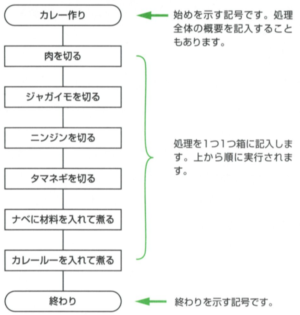
2-9 流れ図はプログラムを考えるための道具

プログラムの流れを図式化した表現手法を、流れ図といいます。流れ図は、記号を線で結んで、プログラムの流れが直感的に理解できるように考えられています。記号はJIS規格で決められていますので、それに従って書けば、誰もが理解できる図になります。

●順次構造を流れ図で表現すると

前節の「カレー作り手順書」を流れ図に書き換えたものが下図です。

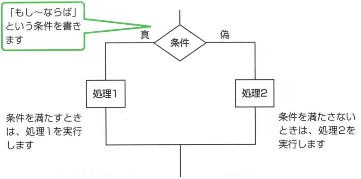
▼カレー作りの流れ図



●選択構造を流れ図で表現すると

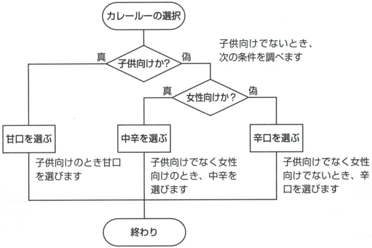
選択構造を流れ図で表すには、以下のように書きます。

▼選択構造の基本図



前節のカレールーを選ぶ場合には、三者択一ですが、これは、下図のように「偽」のとき、次の条件を判定することで実現します。

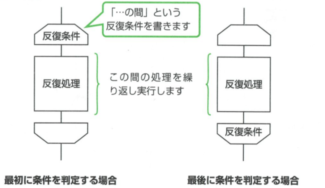
▼カレールー選びの選択構造



●反復構造を流れ図で表現すると

反復構造を流れ図で表すには、以下のように書きます。

▼ 反復構造の基本図



左側の図は、反復処理を行うか、それともループを終了して次の処理に移るかどうかの判定を、反復処理の前に行う場合です。ですから、最初から反復条件を満たさないときには、反復処理を一度も実行しないでループを終了することがあります。たとえば、カレーにジャガイモを入れない場合、ジャガイモ0個としておけば、「ジャガイモを切る」と手順書に書かれていても、何もしないことになります。

右側の図は、反復処理が終わったあとで次にもう1度繰り返すかどうかを判定します。ですから、最低1回は反復処理を行うことになります。両者の性質をよく理解して、使い分けることが大切です。

One Point 用語解説

➡真と偽

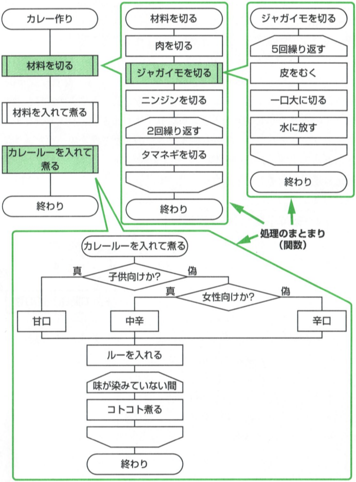
○×クイズでは、答えは「はい」か「いいえ」かのどちらかになります。選択構造でも、反復構造でも、条件には、「子供向けですか?」とか「ジャガイモはまだありますか?」といった、「はい」か「いいえ」で答える問いを書きます。このような式を論理式といいます。

コンピュータ用語では、条件を満たす、つまり「はい」のときを「真」またはTrue、条件を満たさない、つまり「いいえ」のときを「偽」またはFalseと書きます。この2つを論理値といいます。

●関数を流れ図で表現すると

処理のまとまり・関数は、1つの枠に機能の概要のみを書きます。各関数の処理の詳細は、それぞれ別の流れ図にします。「カレー作り」の手順を、流れ図で表現すると下図のようになります。

▼カレー作りの詳細手順の流れ図

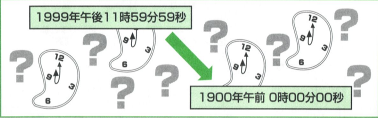


COLUMN

デバッグ

プログラムを作成したら、正しく動作するかどうか検査をします。そして、もし、誤りがあった場合には、プログラムを正しく修正しなければなりません。この作業をデバッグといいます。

デバッグは、プログラムの設計や製造以上に大切な工程です。どんなに綿密に設計しても、どんなに慎重にプログラムを作っても、プログラムの誤り・バグは発生するものなのです。プログラムが大規模になればなるほど、時間をかけてあらゆるケースをテストする必要があります。中には、ユーザが設計者の予想もしない使い方をする場合もありますし、リアルタイム・マルチタスクシステムでは、微妙なタイミングのずれで思わぬバグが発生することもあるのです。2000年問題は、まだ記憶に新しいところですが、設計の見込み違いによるものでした。



実際の業務では、カットオーバーの日取りや、納期が決まっており、さらに、プログラム製造の遅れや仕様変更など、プログラムの検査にかけられる時間はどんどん削られていくことになってしまいます。けれども、コンピュータが生活に深く関わっている今日では、バグのあるプログラムが大きな社会問題を引き起こすことは避けられず、プログラマは、誤りのないプログラムの開発を求められています。

COLUMN

3章

C言語を使ってみよう

これまでは、プログラミング全般の話題でした。第3章では、いよいよC言語を使ったプログラミングの扉を開いてみましょう。そのためには、まず、C言語が使える環境を整えなければなりません。Cコンパイラの入手から、インストール、環境設定を経て、とにかくプログラムを作り、動かしてみましょう。

3-1 C言語の生い立ち

C言語は、もともとオペレーティングシステムを開発するために設計された言語でした。その後広く普及し、国際的に規格化されました。今や、C言語は、最もメジャーなプログラミング言語の１つとしての地位を確立しています。

●誕生

C言語は、1972年、アメリカAT＆Tベル研究所のDennis M.Ritchieによって設計されました。ミニコンピュータのオペレーティングシステムであるUNIXを開発するにあたり、それにふさわしいプログラミング言語が必要だったのです。それより少し前、BCPLという言語を改良したB言語が開発されていました。B言語の次に開発されたので“C言語”と名づけられたといわれています。

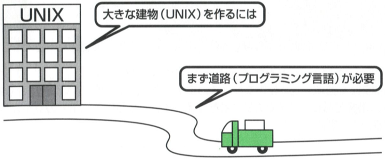
●バイブル

開発者のRitchieは、Brian W.Kernighanと共同で、C言語の仕様をまとめた『The C Programming Language』を出版しました。この本は、著者の頭文字をとって「K&R」と呼ばれ、長い間、C言語の唯一の教科書として、また、規格書として多くのプログラマに読まれてきました。日本でも『プログラミング言語C』という書名で翻訳出版されています。

●多様化

初めは、オペレーティングシステム開発用として生まれた言語でしたが、その優れた言語仕様のため、ミニコンピュータのみならず、パソコンから大型コンピュータまで、また、OSのみならず、アプリケーションや組み込みソフトウェアの開発にも用いられるようになりました。これに伴ってCコンパイラも多数開発されました。ところが、あるコンパイラには新しい機能が盛り込まれてい

▼C言語の誕生

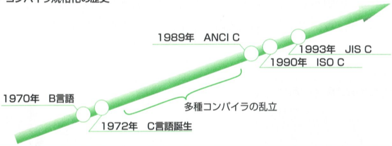


たり、また、もともとK&Rであいまいだった点がコンパイラごとに異なっていたりと、コンパイラによって細かい文法のズレが生じるようになってきました。

●規格化

こうした混乱を経て、1989年末、ANSI（American National Standard Institute：米国国内規格協会）によって、規格がまとめられました。さらに、1990年には、ISO（International Standard Organization：国際標準化機構）規格が、1993年には、日本でもそれを受けてJIS Cが制定され、Cコンパイラは規格化されました。

▼コンパイラ規格化の歴史



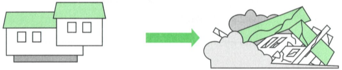
3-2 C言語は構造化プログラミング向き

C言語は、オペレーティングシステムからアプリケーションまで、どんなソフトウェアの開発にも適応できる特徴を持っています。また、プログラミング言語としては比較的後発であり、今までの反省を踏まえ、すっきりとしたプログラムが書けるようになっています。

●歴史的な反省

ソフトウェアの開発は、ハードウェアのそれに比べ柔軟性に富んでいるため、開発者は、綿密な設計をしなくてもプログラムを書くことができてしまいます。その手軽さの反面、継ぎはぎだらけのプログラムは、バグの温床となります。重大なバグが潜んだまま使われていたソフトウェアのために、あるとき不具合が起きて社会問題になったり、バグを治そうとして新たなバグを発生させてしまうなど、過去に作られたプログラムの修正や改版に多くの技術者のパワーが割かれる事態に陥ってしまいました。また、一度作ったプログラムがそのまま永久に使われ続ける、ということはまずありません。社会の変化やニーズの多様化に合わせ、改版の必要が生じてくるのです。そのとき、最初にプログラムを開発した技術者が改版を担当できない場合もありますし、運よく担当できたとしても、時間が経てば忘れていることもあるでしょう。以上のような背景から、バグが潜みにくく、容易に改版しやすいプログラムであることの重要性が叫ばれるようになりました。

▼プログラム設計を建築にたとえる



土台のしっかりしていない家は無秩序に建て増しすると…崩れる

●構造化プログラミング

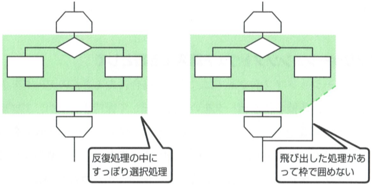
そこで提案されたのが、構造化プログラミングと呼ばれるプログラミングスタイルです。構造化プログラミングは以下の2つの考え方で構成されています。

1. すべてのプログラムは3つの基本構造の組み合わせで書ける
2. 部品化することで見通しのよいプログラムが書ける

すなわち、処理のまとまりを部品化し、その1つ1つは3つの基本構造、順次構造・選択構造・反復構造の組み合わせで記述するというものです。

C言語の言語仕様は、構造化プログラミングがしやすいように設計されています。

▼基本構造の組み合わせ



(a) 構造化プログラミングの例　(b) 構造化違反の例
反復処理の中に選択処理がある場合

3-3 C言語は万能な優等生

C言語は、もともとオペレーティングシステム開発用に設計された言語であるため、高級言語でありながら、コンピュータのハードウェアに直結した部分をも書くことができます。一方で、アプリケーション開発や組み込みソフトウェア開発にも対応できる言語仕様を持っています。

●オペレーティグシステム開発用言語として

オペレーティングシステムは、ハードウェアとアプリケーションソフトウェアの間に立つソフトウェアですから、ハードウェアの構成に関わる部分の記述が必要になります。それは、従来はアセンブリ言語でないと書けないことでした。C言語は高級言語でありながら、“低水準な”言語としての横顔を持っています。

●アプリケーションソフトウェア開発用言語として

アプリケーションソフトウェアは、多くの場合、たくさんのデータを扱います。ですから、多種多様なデータをいかに整理して保持するかが、アプリケーション開発の重要なポイントの1つになります。規定したデータ構造を、そのままプログラムで表現できるよう、C言語には、基本のデータ型のほかに、配列・ポインタ・構造体・共用体というデータ構造が用意されています。

また、処理のまとまりを関数で記述することで、解決すべき問題を小さく分け部品化することができます。さらに、入出力や文字の扱いなどまでをも、言語仕様に取り込まず、関数の形で供給します。これにより、たとえ環境が異なったとしても、その環境に合った関数に差し替えさえすれば、プログラム部分を変更することなくコンパイルしなおすだけで対応することができるのです。言語仕様が少ない分、覚えなければならない文法が少なくてすみますし、コンパイラも小さくてすみます。

▼C言語のパワー



●C言語の特徴

C言語の特徴をまとめておきましょう。

①構造化プログラミング向きの言語：わかりやすいプログラムが書ける

選択・反復構造のための構文が数種類用意されていて、すっきりしたプログラムになります。

②"低水準な"高級言語：ハードウェアに直結したプログラムが書ける

高級言語でありながら、従来はアセンブリ言語でしか書けなかった部分をも記述できます。

③データ型が豊富：データ構造に即したプログラムが書ける

文字・数値などの基本データ型の他に、配列・ポインタ・構造体・共用体という型があり、データをひとまとめにして扱うことができます。

④コンパクトな言語仕様：少ない文法でプログラムが書ける

言語仕様を最小限に抑え、そのかわり関数を提供するという方法を採ることでプログラマの負担を軽減しています。

⑤関数によるモジュール化：機能を関数化したプログラムが書ける

関数を駆使することにより効率よく開発を進めることができます。

⑥高い移植性：異なるハードウェアにも通用するプログラムが書ける

特別にハードウェアに依存している部分以外は、コンパイルしなおすだけで他のハードウェアで実行することができます。

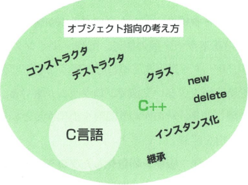
3-4 C言語が進化したC++

現在、開発の現場では、C++がよく使われています。C++は、C言語をベースに、オブジェクト指向の考え方を取り入れた言語です。時代はMS-DOSからWindowsへ。マウスを使ってグラフィカルにパソコンを操作するGUI化の波が押し寄せてきました。それでも、「基本はC言語」なのです。

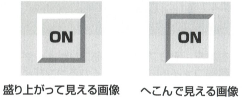
●C++は、C言語の上位互換

オブジェクト指向とは、「データと、それを操作する関数とをワンパックにまとめて扱う」ものです。このまとまりをクラスといいます。クラスは、そのクラスで扱うデータ型の定義と、そのデータを操作する関数の定義とで構成されます。いずれもC言語の文法が基礎となっていますので、C++でプログラミングができるようになるためには、まず第一にCを使いこなせるようにすることが必要です。

▼C言語とC++



▼画面上に描かれるボタンの画像

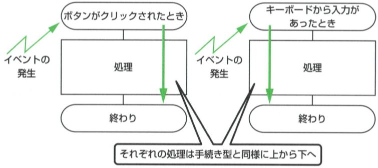


●GUI化の波

Windowsの出現により、アプリケーションソフトウェアは、フレームと呼ばれる窓枠の中で動作するようになりました。ボタンを配してこれをクリックしたり、くぼんだ枠に結果を表示したりと、カラフルな画面をマウスで操作できます。とはいっても、画面上に実際にボタンがあるわけではなく、画面に描画された「ボタンの絵」を見せているにすぎません。クリックした瞬間にボタンがへこんだように見えるのも、その瞬間にへこんだ画像と差し替えているだけなのです。

イベントドリブンのプログラムであっても、各イベントに対する処理の1つ1つは、手続き型と同様、上から下へ実行されます。そういう意味でも、C言語は他のすべての言語の基本といえます。

▼イベントドリブン



3-5 Cコンパイラの準備

C言語でプログラムを書き、そのプログラムをコンピュータで実行するには、ソースプログラムを実行可能ファイルに翻訳してくれるコンパイラが必要です。Cコンパイラを入手し、コンピュータにインストールして開発環境を整えることから始めます。

●エディタの準備

プログラミングをするためには、まず、ソースファイルを入力するためのエディタが必要です。ソースファイルはテキストファイルですから、Windowsに付属しているメモ帳でもできます。WORDや一太郎のようなワープロソフトでもできますが、ファイルを保存するときにワープロ固有のファイル形式ではなく、テキストファイルで、また、ファイル名の拡張子もワープロ固有のものではなく、「.c」となるようにしなければなりません。インターネットのサイトにはフリーのエディタもたくさんありますから、気に入ったソフトをダウンロードして使ってもよいでしょう。

▼メモ帳を使用した例



●Cコンパイラの入手

ソースファイルができたら、次は、それをコンパイルするためのコンパイラが必要です。最近では、コンパイラも無償で手に入るようになりました。ホームページからダウンロードすることもできますし、雑誌や書籍の付録のCD-ROMに収められていることもあります。C++コンパイラでも、C言語のソースファイルをコンパイルすることができます。製品版の中には、エディタとコンパイラ、さらにはデバッグのためのツールからオンラインマニュアルまで、必

要なツールのすべてを統合し、効率よく開発を進められるように整えられている開発環境もあります。

【コンパイラの例】

●無償コンパイラ

LSI C-86 試食版	エル・エス・アイ・ジャパンが販売しているコンパイラの試用版です。試用版とはいえ、C言語の学習をするには十分な機能を備えています。MS-DOS上で動作します。
Borland C++ Compiler	MS-DOS上で動作するC++コンパイラです。C・C++どちらのソースもコンパイルすることができます。また、Windowsアプリケーションを開発することもできます。

●有償コンパイラ

Visual C++	Microsoft社が開発した統合型のコンパイラです。Windowsアプリケーションを自動で生成するウィザードを備えています。
C++ Builder	Borland社が開発した統合型のコンパイラです。GUI開発を効率よく進めることができるRADツールを備えています。

本書では、無償で入手できるBorland C++ Compiler（http://www.borland.co.jp/cppbuilder/freecompiler/）を用いた例を提示していますが、どのコンパイラを使った場合でも基本的な手順は同じです。お使いのコンパイラのマニュアルを参照してください。

●コンパイラのインストールと環境設定

インストールとは、ソフトウェアをパソコンで使える状態にすることです。Visual C++やC++ Builderのような市販コンパイラは、CD-ROMをセットし、簡単な質問に答えるだけで、あとはほとんど自動でインストールしてくれます。が、LSI C-86試食版や、Borland C++ Compilerなどは、手動でインストールしなければなりません。その手順は以下のとおりです。

①圧縮されているファイル群を解凍する

コンパイラはたくさんのファイルで構成されますが、それらのファイルは1つの圧縮ファイルにまとめて提供されます。自己解凍形式であれば、実行することで、そうでなければ圧縮解凍ソフトを使って解凍します。

②決められたフォルダに収める

解凍したファイルは、できるだけマニュアルで勧められている名前のフォルダを作って、そこに収めます。

③ソースプログラムを入れるフォルダを作る

ソースプログラムや、コンパイルしてできた実行可能ファイルを格納するために、コンパイラを収めたフォルダとは別の専用フォルダを用意します。

④PATHを設定する

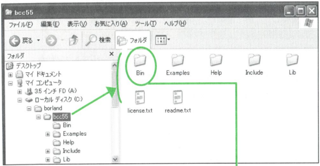
MS-DOS上で作業するとき、カレントディレクトリ（現在の処理対象のディレクトリ）をソースファイルのあるフォルダにしておくと便利です。しかし、それでは、MS-DOSにはコンパイラがどこにあるのかわかりません。そこで、MS-DOSの環境変数PATHにコンパイラが存在するフォルダを追加します。これには、システム設定メニュー（Windowsのバージョンによって呼び名が異なる）の中の［環境変数］画面で設定するか、`C:¥`（Cドライブのルートディレクトリ）にある`AUTOEXEC.BAT`を編集します（次ページ参照）。`AUTOEXEC.BAT`は、マシン起動時に一度だけ実行される特別なバッチファイルです。

⑤その他の必要な設定ファイルを作成・修正する

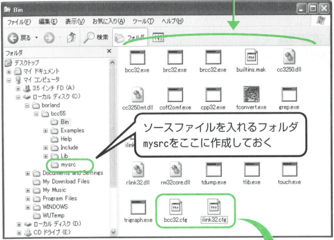
コンパイラごとに必要な設定ファイルをマニュアルに従って修正、または作成し、最後に再起動します（Borland C++ Compilerでは「`.cfg`」という拡張子のテキストファイルを2つ作成）。

【Borland C++ Compilerの場合の環境設定】

▼C:¥borland¥bcc55にセットアップされた中身



▼2つの環境設定ファイルをbinフォルダ内に収める



環境変数

◎Windows98の場合

エディタで、autoexec.batに次の新規行を追加

`path c:¥borland¥bcc55¥bin;%path%`

◎WindowsME/2000/XPの場合

［環境変数］画面で設定

- 既存PATH行があれば、以下を末尾に編集追加
  `;c:¥borland¥bcc55¥bin`
- 既存PATH行がなければ、新規変数名Pathで以下を作成
  `c:¥borland¥bcc55¥bin`

bcc32.cfg

```
-Ic:¥borland¥bcc55¥include
-Lc:¥borland¥bcc55¥lib
```

ilink32.cfg

```
-Lc:¥borland¥bcc55¥lib
```

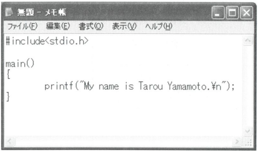
3-6 はじめて書くプログラム

ソースプログラムは、エディタを使って入力します。ソースプログラムを作成し、保存するところまではWindows環境でできますが、これをコンパイルし、実行するのは、MS-DOS上で行います。MS-DOSでは原則としてマウスは使えません。キーボードからコマンドを打ち込む作業になります。

●「私の名前は…」プログラムを書く

初めて書くプログラムは、「自分の名前を画面に表示する」というものです。メモ帳などのエディタを起動して、以下のように打ち込んでください。

▼「私の名前は…」プログラム

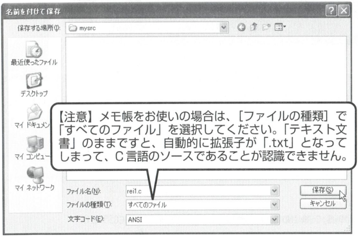


" "で囲まれた"My name is …"の部分は、何でも結構です。この中だけは、かな漢字変換した日本語でもかまいません。ただし、最後に¥nをつけること、" "で囲むことだけは忘れないでください。あとの部分は、そっくりこのとおり、打ち込みます。すべて半角の小文字です。" "以外の部分では、全角の空白があってもいけません。printf(…); の前に空白がありますが、これは、[Tab]キーを使って打ち込むとよいでしょう。

●プログラムを保存する

打ち込みが完了したら、ファイルに保存します。前節で用意したソースを入れるフォルダにrei1.cという名前で保存しましょう。

▼プログラムを保存する



●コマンドプロンプトの起動

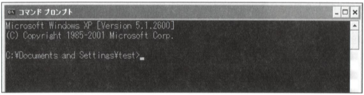
次にコンパイルして実行可能ファイルを作成します。この作業は、コマンドプロンプトを起動して行います。

【コマンドプロンプト（MS-DOSプロンプト）の起動のしかた】

Windows98の場合：	［スタート］ボタン－［プログラム］－［MS-DOSプロンプト］の順に選択して起動
WindowsME/2000の場合：	［スタート］ボタン－［プログラム］－［アクセサリ］－［MS-DOSプロンプト］（ME）［コマンドプロンプト］（2000）の順に選択して起動
WindowsXPの場合：	［スタート］ボタン－［すべてのプログラム］－［アクセサリ］－［コマンドプロンプト］の順に選択して起動

●コマンドプロンプトでソースファイルを確認する

▼MS-DOS画面（コマンドプロンプト）



まず、コンパイラをインストールしたときに設定したPATHが正しく働いているかどうかを確認します。pathと打ち込んでみてください。大文字でも小文字でもかまいません。また、この段階ではプロンプトは状況を問いません。以下はBorland C++ Compilerをインストールした場合の例です。

```
C:¥>path Enter
PATH=C:¥WINDOWS;C:¥WINDOWS¥COMMAND;C:¥;C:¥BORLAND¥BCC55¥BIN

C:¥>
```

表示された情報の中に、コンパイラをインストールしたフォルダが含まれていれば成功です

今度は、先ほど打ち込んだソースファイルが正しく存在することを確認しましょう。cdに続いて、空白とソースプログラムを保存したフォルダの絶対パスを打ち込んでください。

```
C:¥>cd c:¥borland¥bcc55¥mysrc Enter

C:¥borland¥bcc55¥mysrc>
```

プロンプトが、ソースプログラムを保存したフォルダ名に変わります

次にdirと打ち込んで、先ほど作成したソースファイル名を表示させましょう。

```
C:¥borland¥bcc55¥mysrc>dir Enter

ドライブ C: のボリュームラベルはありません.
ボリュームシリアル番号は 07D1-0413
ディレクトリは C:¥borland¥bcc55¥mysrc

.              <DIR>        01-06-29    9:19 .
..             <DIR>        01-06-29    9:19 ..
REI1     C                   63  01-06-29    9:32 rei1.c
               1 個                    63 バイトのファイルがあります.
             2 ディレクトリ    15,718.14 メガバイトの空きがあります.

C:¥borland¥bcc55¥mysrc>
```

dirは、プロンプトに表示されているフォルダにどんなファイルが保存されているかを表示させるコマンドで、Windowsのエクスプローラに似た働きをします。

また、プロンプトに表示されているフォルダは、カレントディレクトリといい、作業の対象となるフォルダです。cdは、作業対象となるフォルダを切り替えるコマンドだったのです。カレントディレクトリをソースファイルが存在するフォルダに切り替えておくと、作業をスムーズに進めることができます。

One Point 用語解説

➡環境変数PATH

MS-DOSは、状態を覚えておくための環境変数と呼ばれる特別な領域をいくつか用意しています。PATHはその1つで、実行可能ファイルを実行しようとしたとき、指定の実行可能ファイルをどのフォルダの中から探し出せばよいか、を登録しておく環境変数です。実行可能ファイルを実行しようとしたとき、MS-DOSは、まず初めにカレントディレクトリを探します。そこに存在しなければ、PATHに登録されているフォルダの中を探すのです。

ですから、コンパイルをしようとしたとき、コンパイラが存在するフォルダがPATHに登録されていないと、MS-DOSは、コンパイラを探し出して実行することができません。

コンパイラのあるフォルダをカレントディレクトリにしてしまえば、問題は解決する、と思われるかもしれませんが、普通は、ソースプログラム専用のフォルダを作り、それをカレントディレクトリにします。実行可能ファイルが存在する絶対パスを毎回毎回指定する方法もありますが、それはとても面倒です。

3-7 コンパイルそして実行

ソースファイルをコンパイルして、実行可能ファイルを生成します。入力ミスなどがあって正しくコンパイルできなかった場合には、実行可能ファイルは作られず、実行することができません。ソースファイルの誤りを訂正し、コンパイルし直す、という地道な作業が続きます。

●コンパイルして実行可能ファイルを作る

コンパイルは、コマンドプロンプト上で

　　　　コンパイラ名　ソースファイル名

というコマンドを実行して行います。Borland C++ Compilerをお使いの場合は、下図のようになります。

コンパイラ名　ソースファイル名

```
C:¥borland¥bcc5¥mysrc>bcc32 rei1.c  Enter
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
rei1.c:
警告 W8070 rei1.c 6: 関数は値を返すべき (関数 main )
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

C:¥borland¥bcc5¥mysrc>
```

【注意】「警告 W8070 rei1.c 6: 関数は値を返すべき(関数 main)」が表示されますが、この警告についてだけは、当面気にしないことにします。LSI C-86コンパイラを使った場合には、この警告は出ません。

●実行可能ファイルの確認

dirコマンドを使ってカレントディレクトリ内のファイルの様子を調べてみましょう。

```
C:¥borland¥bcc5¥mysrc>dir Enter

 ドライブ C: のボリュームラベルはありません.
 ボリュームシリアル番号は 07D1-0413
 ディレクトリは C:¥borland¥bcc55¥mySrc

.        <DIR>                  01-06-29  9:19 .
..       <DIR>                  01-06-29  9:19 ..
REI1     C               63     01-06-29  9:32 rei1.c
REI1     OBJ             555    01-06-29 17:26 rei1.obj
REI1     TDS             393,216 01-06-29 17:26 rei1.tds
REI1     EXE             52,736 01-06-29 17:26 rei1.exe
         4 個      446,570 バイトのファイルがあります.
         2 ディレクトリ   15,314.42       メガバイトの空きがあります.

C:¥borland¥bcc5¥mysrc>
```

コンパイルをする前よりファイルが増えています。それぞれのファイルは以下のとおりです。

▼生成されたファイル

ファイル名	ファイルの種類	説明
Rei1.c	ソースファイル	プログラムを記述したテキストファイル
Rei1.obj	オブジェクトファイル	コンパイルの過程で発生する中間ファイル
Rei1.tds	シンボルテーブル	デバッグに役立つ情報を持ったファイル
Rei1.exe	実行可能ファイル	これから実行するファイル

●プログラムの実行

実行可能ファイル名を入力すると、何が表示されたでしょうか？

```
C:¥borland¥bcc5¥mysrc>rei1.exe Enter
My name is Tarou Yamamoto.

C:¥borland¥bcc5¥mysrc>
```

●うまくいかなかった方へ

「My name is …」という表示が出なかったとしたら、その原因はいくつか考えられます。

・コンパイラのインストールに失敗している

【現象】bcc32 rei1.cと入力したら「コマンドまたはファイル名が違います」と出た

Borland C++ Compilerの場合、エクスプローラで C:¥borland¥bcc55の内容を確認してみてください。各フォルダにファイルが保存されていない場合は、一度フォルダをすべて削除して、解凍からやり直します。

・コンパイラの設定に失敗している

【現象】bcc32 rei1.cと入力したら「コマンドまたはファイル名が違います」と出た

環境変数PATHにコンパイラが存在するフォルダが正しく設定されていますか? コマンドプロンプトでPATHコマンドを実行してみてください。Borland C++ Compiler の場合は、C:¥borland¥bcc55¥binが含まれていない場合は、Autoexec.batファイルを見直します。パスとパスの区切りは、「;」(セミコロン) になっていますか?

その他、コンパイラにとって必要なファイルは、正しく保存されていますか? Borland C++ Compilerの場合には、bcc32.cfgとilink32.cfgが、

c:¥borland¥bcc55¥bin

に必要です。ファイルの詳細は、コンパイラ付属のマニュアルを参照してください。

・ソースプログラムの入力に失敗している

【現象】コンパイラの実行は始まったが、各種エラーや警告が表示された

```
C:¥borland¥bcc5¥mysrc>bcc32 rei1.c
Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland
rei1.c:
エラー E2379 rei1.c 6: ステートメントにセミコロン(;)がない(関数 main )
警告 W8070 rei1.c 6: 関数は値を返すべき(関数 main )
*** 1 errors in Compile ***

C:¥borland¥bcc5¥mysrc>
```

ソースプログラムは、正しく入力できていますか? 大文字と小文字はまちがっていませんか? 「;」を忘れていませんか? エディタで確認してみてください。

・**ソースプログラムの保存に失敗している**

【現象】bcc32 rei1.cと入力したら以下のようなエラーが出た

```
C:¥borland¥bcc5¥mysrc>bcc32 rei1.c
Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland
エラー E2194: ファイル 'rei1.c' が見つからない

C:¥borland¥bcc5¥mysrc>
```

ソースファイル用のフォルダの中に、rei1.cという名前のファイルが保存できていますか? また、カレントディレクトリは正しくソースファイル用のフォルダに切り替えられていますか? コマンド(MS-DOS)のプロンプトで確認してください。

・**コンパイラの起動に失敗している**

【現象】上と同じエラーが出た

コンパイラ名やファイル名を正しく「bcc32 rei1.c」と入力しましたか? MS-DOSは大文字と小文字を区別しませんので、大文字・小文字どちらでもかまいません。もう一度注意して入力してみてください。

3-8 Cプログラムの形式

Ｃ言語は原則としてフリーフォーマットです。何をどこに書かなければならない、という規則はほとんどありません。それでも、最低限の枠組みは必要です。また、自由な形式であるがゆえに、かえってわかりやすく読みやすいプログラムを書くための工夫が求められます。

●Ｃプログラムの枠組み

Ｃプログラムでは、以下のように書きます。

```
#include <stdio.h>
main()
{
                    処理
}
```

・#include <stdio.h>

プログラムの一番先頭には、この一文を書きます。これが何か、ということは後で明らかにすることにして、ここでは、これがないとコンパイルが正常に終了できない、ということだけにとどめます。

・main()

プログラムの始まりを示します。どんなに大きなプログラムにも、どんなに小さなプログラムにも、必ず1つだけ存在しなければなりません。Borland C++ Compilerでは、

```
void main(void)
```

と書くことを推奨しています。それで、コンパイルのときに前節のような警告が出たのですが、ほかのコンパイラでは、main()という記述で警告を発生しないことの方が多いようです。

・{ }

この括弧で囲まれた部分に「コンピュータに何をさせたいのか」という処理を書きます。

・処理

画面に表示する、キーボードから入力する、計算する、など具体的なコンピュータに対する命令を書きます。

改めて、rei1.cのプログラムを見直してみましょう。プログラムは、この形式に沿って書かれていることがわかります。

●コメント文

プログラムは、コンピュータに対する命令の集まりです。高級言語といえども、通常の日本語とは異なる、記号の羅列にすぎません。このプログラムは何をするものなのかという説明がプログラムと一緒に書かれていると、より早く、より深く処理内容を理解することができます。このような説明文をコメント文といい、/*と*/とで囲って書きます。コメント文は、単なる説明文ですから、文法上はどこに何を書いてもまったく支障がありません。コンパイラはコメント文をスキップし、この部分は翻訳しないからです。けれども、特にチームでプログラムを開発している場合などは、ある程度のルールを決めておくとよいでしょう。

プログラムの機能を説明

```
/*********************************************************
        名前を表示するプログラム
*********************************************************/
#include <stdio.h>
main()
{
        printf("My name is Tarou Yamamoto.¥n");  /*名前の表示*/
}
```

個々の命令の説明

3-9 画面に情報を表示する

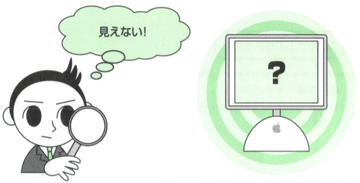
どんなに有益な情報があったとしても、それが蔵の奥深くにしまいこまれていたとしたら、何の役にも立ちません。コンピュータでも同じことです。コンピュータ内部にある有効な情報は、内部に「ある」だけではダメ。ヒトの目に触れて初めて有益なのです。

●情報を画面に表示する文、printf

プログラムを実行するとき、実行に先立って、実行可能ファイルはすべてメモリにロードされます。コンピュータに対する命令だけではなく、データなどもすべて、です。ですから、情報はすべてメモリ上にあります。けれども私たちは、メモリの中身をのぞいて見ることができません。それどころか、メモリのどこ（何番地）にあるのかさえわかりません。このままでは「宝の持ち腐れ」です。

これらの情報を人間に見せる、最も簡単な方法は、画面に表示することです。決まった文字を画面に表示するには、次のように書きます（次ページ）。

▼私たちはコンピュータの中身を肉眼で見ることはできない！



```
printf("　表示したい文字　");
```

P.81のプログラムでは、「My name is Tarou Yamamoto.」と表示しました。"と"で囲まれた部分は、かな漢字でもかまいません。

```
printf("私の名前は山本太郎です¥n");
```

のように書くこともできます。が、それ以外の部分は、半角英数文字でなければなりません。特に ( ) や " などは全角になっていても気づかないこともあるので、ご注意ください。

●拡張表記（エスケープシーケンス）

printf(…); の文で、表示したい文字の最後につけた「¥n」は、改行を表す記号です。このように、「¥」と組み合わせて特別な意味を表す文字を拡張表記（エスケープシーケンス）といいます。見た目は、2文字のように見えますが、「¥」と「n」の2つで1つの記号を表します。JIS規格では、¥ではなく、\（バックスラッシュ）と規定されていますが、国内のパソコンには\のキーはないので、通常は¥で代用します。¥n以外にも、下表のようなものがあります。

▼拡張表記のいろいろ

表記	意味	表記	意味
¥'	文字 '	¥n	改行（次行の先頭へ）
¥"	文字 "	¥r	復帰（現在の行の先頭へ）
¥?	文字 ?	¥t	水平タブ（次の水平タブ位置へ）
¥¥	文字 ¥	¥v	垂直タブ（次の垂直タブ位置へ）
¥a	警報	¥ooo	8進数（oooは0～7）
¥b	後退	¥xhh	16進数（hhは0～F）
¥f	書式送り（次ページの先頭へ）		

P.81の前出プログラムは、正確には、「My name is Tarou Yamamoto.」と表示したあと改行する、というものだったのです。

COLUMN

カレントフォルダの自動切り替え

コマンドプロンプトを起動したあと、カレントディレクトリをソースファイルのあるフォルダに変更するために、毎回、cd c:¥borland¥bcc55¥mysrc と入力しなければなりません。毎回のことですから、これを起動のたびに自動的に行う方法をご紹介しましょう。

①プロンプトのショートカットの作成

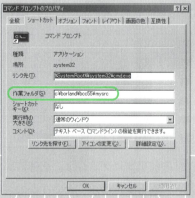
デスクトップに、コマンドプロンプトまたはMS-DOSプロンプトのショートカットを作成します。

1）［スタート］ボタンから［プログラム］メニュー－［アクセサリ］メニューを表示します（WindowsXPの場合）。

2）［コマンドプロンプト］のメニューを右クリックで選択したまま、デスクトップまでドラッグします。その状態で指を離して「コピーする」を選択すると、デスクトップにコマンドプロンプトのショートカットが作成されます。

② カレントフォルダの登録

作成したショートカットアイコンを右クリックして、［プロパティ］を選択します。［ショートカット］タブに［作業フォルダ］という項目があります。ここに、
c:¥borland¥bcc55¥mysrc
と登録しておくと、このショートカットをダブルクリックして起動したとき、自動的にカレントフォルダが切り替わります。



この章からは、本格的にプログラミングをしていきます。プログラムは、「入力された情報を加工し、結果を出力する」のが基本です。ここでは、これからの学習のすべての基礎となる、

情報の器を用意して演算してみよう

4 章

入力・出力・演算、そして、情報を記憶しておくための方法を学びます。プログラミングの学習の最短距離は、とにかくやってみることです。前章でインストールした環境を使って、経験を積んでいきましょう。

4-1 データを型に分類する

C言語は、すべてのデータを4つの「型」、①符号つき整数型、②符号なし整数型、③浮動小数点型、④文字型、に分類して扱います。型によって、コンピュータ内部の表現の方法が異なります。1つのデータが占める領域の大きさ（バイト数）により、もっと細かく分類します。

●なぜ型による分類が必要なのか

コンピュータの内部では、情報は、すべて2進数、0と1とで表現されることは、すでに学習しました。それでは、たとえば、

0000 0000 0011 0111

というデータがあったとき、これはいったい何を表しているものなのでしょうか？　これを数値の2進数と考えて、10進数に変換すると、

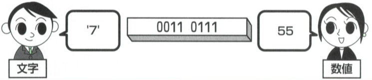
$$2^0 \times 1 + 2^1 \times 1 + 2^2 \times 1 + 2^3 \times 0 + 2^4 \times 1 + 2^5 \times 1 = 55$$

となります。ところが、これが文字コードだとすると、第1章P.17のASCIIコード表から、文字の'7'となり、異なる結果となります。また、

0000 0001 0000 0001

というデータがあったとき、16ビット全部をひとかたまりと考えると、10進数では「257」になりますが、8ビットずつ2つのデータがある、と考えると、「1」が2個ということになります。広いメモリを、どのように区切り、そこにどのような形式のデータが入っているのかをあらかじめ決めておかなければ、'0'と'1'を正しいデータとして認識することができないのです。

▼同じデータでも、型によって意味が異なる



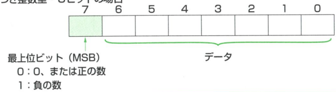
●符号つき整数型（int型）

負の数を含む整数を扱います。2進数で表現しますが、最上位ビット（MSB：Most Significant Bit）は、符号を表すのに使われます。8ビットでは、

$$-2^7 \sim 2^7-1$$

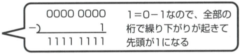
を表現できます。

▼符号つき整数型・8ビットの場合



例

10進数	2進数
12	0000 1100
-1	1111 1111



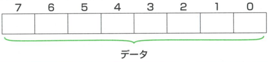
●符号なし整数型（unsigned int型）

すべてのビットを正のデータを表すために使います。8ビットでは、

$$0 \sim 2^8-1$$

を表現できます。

▼符号なし整数型・8ビットの場合



例

10進数	2進数	16進数
12	0000 1100	0x0c

数値の先頭に0x（または0X）をつけて、16進数であることを表します。

●浮動小数点型

数値の部分（可数部）と10のべき乗の部分（指数部）とに分けて表現します。

例　132.5　　1.325×10^{2}

　　0.0035　　3.5×10^{-3}

▼浮動小数点型の内部表現の例（132.5）



（注意）浮動小数点数をコンピュータ内部でどう表現するかについては、規定がなく、それぞれの処理系に任されています。

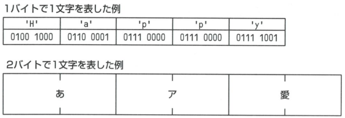
●文字型

ASCIIコードとみなす型で、1バイト（8ビット）を1つの単位として扱います。ソースプログラム上で、文字データを表すときには、'と'で囲んで表します。

例　'a'　'A'

日本語の漢字やひらがなのように、2バイトを1組として1文字を表す2バイト文字も許されています。

▼1バイト文字と2バイト文字



●文字列

文字は、1つ1つ独立に扱うこともありますが、何文字かまとまって意味を持つこともあります。このひと続きの文字を文字列と呼びます。文字列は、文字を並べ、その最後に「ここで終わり」という印の文字を並べます。終わりの印は、

0000 0000

であり、ソースプログラム上では'¥0'と書きます。これは、ヌル文字（null）と呼ばれます。

▼末尾にヌル文字のついた文字列の例

'H'	'a'	'p'	'p'	'y'	'¥0'
0100 1000	0110 0001	0111 0000	0111 0000	0111 1001	0000 0000

ひと続きの文字列をソースプログラム上で書くときは、全部の文字を"と"で囲って書きます。

例　"Happy"

ですから、1と'1'と"1"は、すべて違うものを表現します。

1	数値の1	0000 0000 0000 0001
'1'	文字の'1'	0011 0001
"1"	文字列で'1'と'¥0'の続き	0011 0001 | 0000 0000

ソースプログラム上での10（数値の10）や、'1'（文字の1）、"Happy"（文字列）など、直接データを表す表現を**定数**といいます。

4-2 変数を使う

データは、その時々で変化します。毎回同じ処理をする場合でも、扱うデータは異なります。異なったデータに対応するために、データそのものを指定するのではなく、データを入れておく箱を用意します。この箱のことを変数といいます。

●変数を使うと、柔軟性が増す

年齢を表示するプログラムを考えてみましょう。画面に、

```
I am 18 years old.
```

と表示させるプログラムは、以下のようになります。

```
#include <stdio.h>
main()
{
        printf("I am 18 years old. ¥n");
}
```

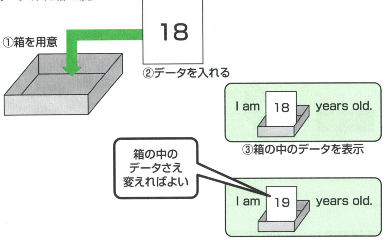
しかし1年経って19歳になったら、プログラムを19に書き換えて保存しコンパイルする、という作業をしなければなりません。

▼プログラムの更新作業



そこで、箱を用意し、なんらかの手段でその箱に「年齢」というデータを入れることができたら、そして、「18」とか「19」というデータそのものではなく、「箱の中の数値」を表示するよう命令することができたら、年齢が変わっても、なんらプログラムを変更することなく処理を行うことができるでしょう。

▼データの入る箱を用意する



●箱（変数）の正体

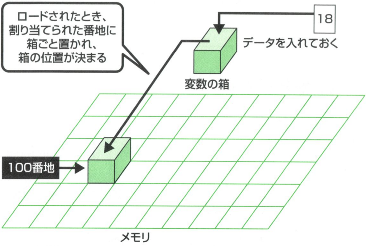
プログラムは、実行に先立って、メモリにロードされる（読み込まれる）のでしたね。データも同様に、メモリにロードされます。箱（変数）を用意する、とは、

広いメモリの中の、ある場所を特定のデータのために確保する

ということです。ところが、それが具体的に何番地になるのかということは、普通はロードされてみなければわかりません。オペレーティング・システムが、そのとき空いている、都合のよいメモリの番地を割り当てるからです。そこで、データをしまう箱に名前をつけ、この名前で管理します。これが**変数名**です。

変数という箱にデータを入れておいて、ロードされたときには、割り当てられた番地にデータが箱ごと置かれる、というイメージになります。

▼ 変数の役割



●箱を用意する：変数の宣言

変数は、あらかじめ宣言し、この変数が、どの型のデータをしまうものなのかを明らかにしなければなりません。

変数の宣言は、

```
型指定子    変数名 = { 初期化子 };
```

の形式で行います。初期化子とは、変数に初期値を与えるときのデータです。

▼ 主な型指定子

型指定子	データの種類	型指定子	データの種類
int	符号付き整数	float	単精度浮動小数点数
long int	符号付き整数	double	倍精度浮動小数点数
unsigned int	符号なし整数	char	文字

【プログラム例】

```
/*****************
        変数の宣言
************************************************/
main()
{
        int     nenrei;  /*int型の変数の宣言*/
        char    seibetu; /*char型の変数の宣言*/
        double  sincyou; /*double型の変数の宣言*/
}
```

プログラム全体の機能を表すコメント文を書いておくとよいでしょう

文の終わりにはセミコロンをつけます

{ と }の間に変数の宣言を書きます

1つ1つの説明をコメント文にしておくとよいでしょう

●箱にデータを入れる（初期化と代入）

変数にデータを入れるには、2つの方法があります。

①初期化

変数の宣言と同時に初期値を設定します。

例
```
int nenrei = 18;            /* 年齢・初期値18 */
char seibetu = 'M';         /* 性別・初期値'M'*/
double sincyou = 179.4;     /*身長・初期値179.4*/
```

②代入

あらかじめ宣言してある変数に、あとで、データを代入することができます。代入することにより初期化されている変数の値を、途中で変更することもできますし、宣言時の初期化を省略し、最初に代入することもできます。

例
```
nenrei = 18;          /*年齢に18を代入*/
seibetu = 'M';        /*性別に文字 M を代入*/
sincyou = 179.4;      /*身長に179.4を代入*/
```

4-3 変数の中の情報を表示する

変数の中には、いったいどんなデータが入っているのか、それを知る最も簡単な方法は、変数の内容を画面に表示してみることです。前出のprintfには、書式を指定する機能があり、データを見やすく編集して表示することができます。データの型によって書式の書き方が少しずつ違います。

●printf文を使って変数の内容を表示する

決まった文字を画面に表示する方法は、

```
printf("    表示したい文字    ");
```

でした。表示したい文字の部分に変数の内容が含まれている場合には、変数の位置に、変数の型に応じた**変換指定子**と呼ばれる記号を書きます。たとえば、int型変数nenreiの内容が18であるとき、これを使って、

```
I am 18 years old.
```

のように表示したいときには、

```
printf("I am %d years old. ¥n",nenrei);
```

と書きます。" "で囲まれた部分を書式といいます。書式の中に書かれている%dは、int型のデータを表す変換指定子で、%dがそのまま表示されるのではなく、%dのかわりに、「,」以降に書かれている変数名nenreiの内容が表示されます。ここには変数名だけではなく、データを直接書くこともできます。

【printf文の一般形】

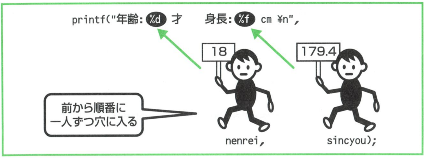
```
printf(" 書式 ", 変数の並び);
```

▼主な変換指定子

変換指定子	型	意味
%d	int	int型のデータを10進数で表示する
%x	int	int型のデータを16進数で表示する
%f	double	double型のデータを小数点付きで表示する
%c	char	1文字表示する
%%		文字 %を表示する

●一度に複数個の変数を表示

1つの文の中に表示したい変数を2つ以上含む場合には、変換指定子を必要な数だけ書き、それが出現する順番に後ろに変数名を書きます。書式に含まれる変換指定子の数と、あとに続く変数の数とは同じでなければなりません。



【プログラム例】

```
printf("年齢: %d 才  身長: %f cm ¥n",nenrei,sincyou);
```

【実行結果】

```
年齢: 18 才  身長: 179.400000
```

※表示桁指定については次ページ参照

4-4 数値データを編集して表示する

いくつかの数値データを1行に1つずつ表示するとき、各数値の桁位置がそろっていると、とても見やすくなります。また、小数の場合には、小数点以下の有効桁数も指定したいものです。Format文の書式では、これらの桁数の指定をすることができます。

●int型データの桁数を指定（フィールド幅）

int型データの桁数の指定とは、表示する桁を確保することです。%とdの間に確保したい桁数を整数で書きます。数値データは、確保された領域内で右詰めに表示されます。

【プログラム例】

```
printf("I am %5d years old. ¥n",nenrei);
```

【実行結果】

```
My name is    18 years o
```

5文字分の表示領域をnenreiのために確保

データの桁数が指定したフィールド幅を超えている場合には、フィールド幅を超えて、すべて表示されます。

【プログラム例】

```
printf("%3d¥n",1000);
```

【実行結果】

```
1000
```

指定は3桁だが、データがはみ出して表示されている

●int型データの最小桁数指定（精度）

最低限表示しなければならない桁数を指定することができます。ピリオド（.）に続けて整数を書きます。データの桁数が足りないときは、左側に0が補われます。

【プログラム例】

```
printf("%8.4d¥n",10);
```

【実行結果】

0010

8桁のフィールド幅のうち、
最低4桁は表示

16進数を表示するときなどによく使われます。

【プログラム例】

```
printf("%8.4x¥n",31);
```

【実行結果】

001f

10進数の20は、16進数では1fとなる

データが0で精度も0のときは、何も表示されません。

【プログラム例】

```
printf("%8.0d¥n",0);
```

【実行結果】

何も表示されない

●double型データの小数点以下の桁指定（精度）

変換指定子が%fのときは、ピリオド（.）に続けて書かれた整数は、小数点以下の桁数を指定します。データは四捨五入されます。

【プログラム例】

```
printf("%.3f¥n",12.34567);
```

【実行結果】

12.346

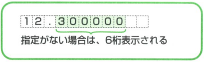
小数点以下第3位で四捨五入した
値が表示される

精度が省略されたときは、6桁となります。

【プログラム例】

```
printf("%f¥n",12.3);
```

【実行結果】

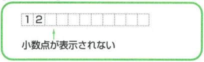


精度が0の場合には、小数点は表示されません。

【プログラム例】

```
printf("%.0f¥n",12.3);
```

【実行結果】



●double型データの桁数を指定（フィールド幅）

フィールド幅には、小数点も小数点以下の桁数も符号も、すべてを含んだ桁数を指定します。

【プログラム例】

```
printf("%8.4f¥n",-12.34567);
```

【実行結果】



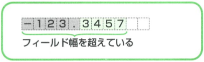
この例の場合には、小数部4桁、小数点に1桁、マイナスの符号に1桁必要ですから、整数部表示の桁数は2桁となります。

整数部の桁数が足りないときは、フィールド幅を超えて表示されます。

【プログラム例】

```
printf("%8.4f¥n",-123.34567);
```

【実行結果】



●左詰め指定

%とフィールド幅指定の間にマイナス記号（-）を書くと、データは、フィールド内で左詰めに表示されます。

【プログラム例】

```
printf("%-4d¥n",10);
```

【実行結果】



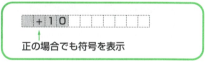
●＋の表示

負の数を表示する場合には、必ず先頭に負の符号（-）がつきますが、%とフィールド幅指定の間に+を書くと、正の場合にも符号（+）を表示します。

【プログラム例】

```
printf("%+4d¥n",10);
```

【実行結果】



●16進数の0xの追加

C言語では、数値が16進数であることを表すのに、先頭に0x（または0X）をつけます。%x（または%X）を使って16進数で表示する際、%とフィールド幅指定の間に#を書くと、先頭に0x（または0X）が補われます。

【プログラム例】

```
printf("%#8.6x¥n",25);
```

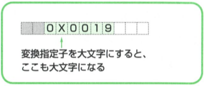
【実行結果】



【プログラム例】

```
printf("%#8.6X¥n",25);
```

【実行結果】



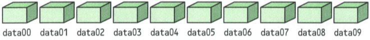
4-5 配列を使って データをまとめて扱う

データがたくさんあるとき、1つのデータについて1つの変数を確保していたのでは、たいへんです。同じ型の同じ種類のデータをまとめて扱えるようにしたものを配列といいます。配列には、用意した要素の数だけのデータを入れることができます。

●配列とは

10個の整数を記憶するには、10個の変数が必要です。すべての変数には異なった名前をつけ、すべて別々に管理しなければなりません。

▼すべての変数は異なった名前でなくてはならない



これらの変数を全部ひとまとめにし、すべてを1つの名前で管理できるようにしたものを**配列**といいます。

▼配列で一元管理

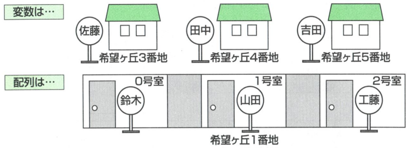


家にたとえると、変数が一戸建て住宅であるのに対し、配列は、マンションのような集合住宅です。一戸建て住宅には1家族しか住むことができませんが、集合住宅には、戸数分だけの家族が住むことができ、それぞれ別々に生活しています。これと同様に、配列には、用意された要素の数のデータを入れることができるのです。そして、一戸建てには一軒一軒に住所があるのに対し、集合住宅は、建物全体に1つの住所が割り当てられています。このままでは、マンションに住む個々の家族を区別することができませんので、それぞれの部屋を識別するために部屋番号を使います。住所にあたるものが、変数名・配列名です。変数は、1つずつ異なる変数名をつけますが、配列は、全部で1つの配列

名をつけます。そして、各要素を識別するために**添え字**と呼ばれる番号を使います。マンションの部屋番号と同じです。

●配列の宣言

▼変数名と配列名



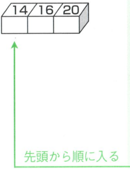
変数と同様、配列を利用する場合には、あらかじめ宣言をしなければなりません。型・配列名のほかに、この配列が何個のデータをしまうことができるかを明らかにします。これを**配列の要素数**といいます。

配列の宣言は、

```
型指定子　　配列名[要素数] = { 初期化子の並び };
```

の形式で行います。

【プログラム例】

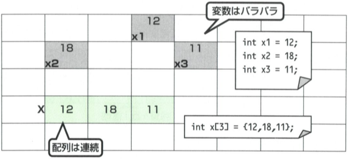



```
/**********************************
          配列の宣言
**********************************/
#include <stdio.h>

main()
{
          /*配列の宣言と初期化*/
          int     data[3] = {14,16,20};
}
```

●メモリに配列を割り当てる

変数も配列も、データを記憶しておく「箱」、つまりメモリ内に確保された領域です。変数は、この箱が1つ1つバラバラになっているため、いくつかの変数を宣言すると、これらの変数が互いにメモリ内のどこに割り当てられるかわかりません。一方、配列はひと続きのデータであり、同じ配列内の要素は連続した番地に割り当てられることが保証されています。



●配列内のデータを指定する

配列は、たくさんのデータを一度に扱うことができますが、配列の中の1つ1つを指定することも必要です。C言語では、先頭を0番目とし、順に、1番目、2番目…という番号で、それぞれのデータを区別します。これを添え字といい、整数を[ ]で囲って書きます。

配列を宣言したときは、要素数を指定しましたから、たとえば**int**型の配列名x、要素数3個であれば

```
int     x[3];
```

のように指定します。各要素を表す添え字は0から始まりますので、

```
x[0], x[1], x[2]
```

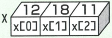
の3個となり、x[3]はありませんのでご注意ください。

【プログラム例】

```
/**********************************
        配列の宣言
**********************************/
#include <stdio.h>

main()
{
        /*配列の宣言*/
        int     data[3] ;

        /*配列へのデータの代入*/
        data[0] = 12;
        data[1] = 18;
        data[2] = 11;
}
```



配列名は、配列全体を表しますが、添え字のついたものは、その中の1つのデータを指し表します。つまり、添え字をつけると、変数とまったく同じように扱うことができるのです。

【プログラム例】

```
/**********************************
        配列の宣言
**********************************/
#include <stdio.h>

main()
{
        /*配列の宣言と初期化*/
        int     data[3] = {12,18,11};

        /*各データを表示*/
        printf("配列data の%2d番目は %4dです¥n",0,data[0]);
        printf("配列data の%2d番目は %4dです¥n",1,data[1]);
        printf("配列data の%2d番目は %4dです¥n",2,data[2]);

}
```

【実行結果】

```
配列data の 0番目は   12です
配列data の 1番目は   18です
配列data の 2番目は   11です
```

4-6 配列を使って文字列を表現する

char型の1つの変数には、1つの文字が入りますから、いくつかの文字のまとまりである文字列を扱うには、char型配列が適しています。char型配列を使うことで、文字列を一括して扱えますし、配列の各要素を指定することで、1文字単位で扱うこともできます。

●配列に文字列をしまう

文字列は、文字の並びと、おしまいを示すヌル文字'¥0'で構成されるものです（P.89参照）。char型配列の先頭から順に文字が入ります。



文字列では、終わりを示すヌル文字'¥0'が必ず最後につきますから、文字列を記憶するのに必要な配列要素数は、文字数＋1となります。上の図の例では、5文字のデータを記憶するために、a[0]～a[5]の6個の要素数を持つchar型配列を用意しています。ここで、配列名はaとしています。

●char型配列を文字列で初期化する

char型配列の宣言は、int型の場合とまったく同様です。

【プログラム例】

```
/****************************************
  char型配列の宣言
****************************************/
#include <stdio.h>

main()
{
  /*配列の宣言と初期化*/
  char    a[6]={'H','a','p','p','y','¥0'};
}
```

【実行結果】

（上図）

初期化したいデータが文字列のとき、つまり、ひと続きの文字の最後に'¥0'がついたデータであるときは、文字列を表す" "を使って下の例のように書くことができます。

【プログラム例】

```
/*******************************************
        文字列の初期化
*******************************************
#include <stdio.h>

main()
{
        /*配列の宣言と初期化*/
        char    a[6] = "Happy";

}
```

このような書き方は、初期化のときにのみ可能です。" "を使って配列に文字列を代入することはできませんので、ご注意ください。

【誤ったプログラム例】

```
/*******************************************
        文字列の代入
*******************************************
#include <stdio.h>

main()
{
        /*配列の宣言と初期化*/
        char    a[6];

        a = "Happy";        ERROR 発生!!
}
```

初めに配列の宣言だけしておいて、あとでデータを代入したいときには、1文字ずつ各要素に代入しなければなりません。

4-7 文字データを編集して表示する

文字データの表示は、基本的には、数値の場合と同様です。表示位置が揃うように書式を指定することなどができます。数値や文字は、1つの変換指定子で1つのデータを表示しますが、文字列の場合は、1つの変換指定子で文字列内のすべての文字を表示することができます。

●文字・文字列を指定する変換指定子

文字データを表示する変換指定子は、以下の2つがあります。

▼変換指定子

変換指定子	型	意味
%c	char	1文字を表示する
%s	char	文字列を表示する

%cがたったの1文字を表示するのに対し、%sは、文字列をまとめて扱います。

【プログラム例】

```
/***********************************
        文字と文字列の表示
***********************************/
#include <stdio.h>

main()
{
        /*文字列の宣言と初期化*/
        char    a[6] = "Happy";

        printf("1文字目は       %c¥n",a[0]);
        printf("文字列全部では %s¥n",a);
}
```

1文字だけを表示するので、添え字をつけて指定

文字列全部を表示するので、配列名だけでよい

【実行結果】

```
1文字目は        H
文字列全部では Happy
```

'H'	'a'	'p'	'p'	'y'	'¥0'
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

●char型データの桁数を指定（フィールド幅）

文字・文字列を表示する領域を確保します。文字列では、文字数に対してフィールド幅が足りないときは、フィールド幅を超えて、すべて表示されます。

【プログラム例】

```
printf("%3c¥n",'M');

printf("%3s¥n","Happy");
```

【実行結果】

```
  M
```

3桁確保し右詰め

```
Happy
```

3桁を超えてすべて表示

●文字列の最大文字数（精度）

変換指定子が%sのときは、ピリオド（.）に続けて書かれた整数は、表示する最大の文字数を指定します。文字列が指定された最大文字数より多いときは、後ろが切り捨てられます。

【プログラム例】

```
printf("%8.3s¥n","Happy");
```

【実行結果】

```
     Hap
```

8桁確保した領域に、3文字だけ右詰めで表示

●左詰め指定

%とフィールド幅指定の間にマイナス記号（-）を書くと、データは、フィールド内で左詰めに表示されます。

【プログラム例】

```
printf("%-8s¥n","Happy");
```

【実行結果】

```
Happy
```

左詰めで表示

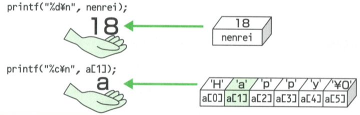
4-8 配列名とアドレスとprintf

配列名は、配列の名前を表すと同時に、もう1つの役目があります。それは、配列としてメモリ内に確保されたひと続きのデータの先頭の番地（アドレス）を示す、という役目です。これに対し、変数名は、中に入っているデータを表します。

●printfの変換指定子が欲しいもの

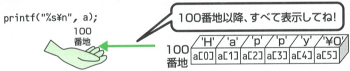
printf文の変換指定子は、表示の際にデータに置き換わります。%dや%f、%cなど、1つのデータを表示する変換指定子には、表示するデータそのものが渡されます。ですから、カンマ（,）の後ろには、変数名や配列名に添え字をつけて1つの要素を指定したもの（データを表す）を書くのです。

▼printfの変換指定子にデータを渡す



文字列を表示する%sの場合はどうでしょうか？　文字列は、文字をまとめて扱いますから、一度にたくさんのデータが必要になります。そこで、%sには、データではなく、データが記録されている先頭の番地（アドレス）を渡します。配列の各要素は、メモリ内では、連続した番地にあることが保証されていますので、先頭の番地さえ渡しておけば、あとは次々とデータを得ることができるのです。配列名は先頭のアドレスを表しますから、カンマ（,）の後ろには配列名を書きます。

▼配列から、printfの変換指定子にデータを渡す場合



●%dと%sの違い

それでは、変換指定子%dに配列名を渡すと何が表示されるのでしょうか？%dは、受け取ったものをそのまま表示します。配列名は、先頭のアドレスを表しますから、そのとき、その配列がロードされているメモリ内のアドレスが表示されるのです。

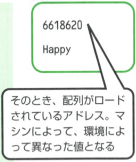
【プログラム例】

```
/************************************
        %dと%sの違い
************************************/
#include <stdio.h>

main()
{
        /*文字列の定義と初期化*/
        char    a[6] = "Happy";

        printf("%d¥n",a);           /*アドレスの表示*/
        printf("%s¥n",a);           /*文字列の表示*/
}
```

【実行結果】



配列名という箱（アドレス）をもらったとき、%sは、箱を開けて中の文字列を表示しますが、%dは、箱（アドレス）をそのまま表示するのです。

▼%sと%d



4-9 算術演算子とデータの型

数値に対する四則演算を行う演算子を算術演算子といいます。同じ型同士で演算を行った場合には、結果もその型になります。型が異なるデータ同士で演算を行う場合、精度の高い方の型に揃えてから、演算を行います。結果も、精度の高い方の型になります。

●算術演算子の種類

数値に対する算術演算は、下表のとおりです。

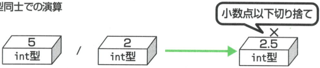
▼算術演算子

算術演算子	演算の種類	備考
+	加算	
−	減算	
*	乗算	
/	除算	割る数が0であってはならない
%	剰余	割る数が0であってはならない。整数型のみ

●除算の結果と型

int型同士で演算を行うと、結果も**int**型となりますが、除算の場合は、**int**型÷**int**型が必ずしも**int**型にはなりません。得られた商の小数点以下は、切り捨てられてしまいます。

▼int型同士での演算



どちらか一方でも**double**型であれば、**double**型の結果を得ることができます。

▼double型の結果を得る場合



●キャスト演算子

int型変数 ÷ int型変数を行うと、結果がint型になってしまいます。演算を行う一瞬の間だけ、本来の型と異なる型に変身させる演算子をキャスト演算子といい、以下の形式で書きます。

```
(型) 変数名
```

▼キャストの役割



【プログラム例】

```
/*************************************
        キャスト演算子
*************************************/
#include <stdio.h>

main()
{
        int     x=5;
        int     y=2;

        printf("x/yの結果は %f ¥n",(double)x / y);
        printf("x/yの結果は %d ¥n",(double)x / y);
}
```

【実行結果】

```
x/yの結果は 2.500000
x/yの結果は 0
```

結果はdouble型で得られるが、変換指定子が%dで、両者が一致しないため、正しく表示されない

4-10 代入演算子

C言語では、右辺と左辺とを結ぶ = は、右辺のデータを左辺の変数に代入する、という意味で、数学の = の意味とは異なり、右辺と左辺とが等しいわけではありません。両辺の型が一致していないときは、左辺の型に変換されてから代入が行われます。

●単純代入演算子で代入する

左辺には、右辺のデータを受け取る変数名や添え字つきの配列名を書きます。左辺に演算子を書くことはできません。

【プログラム例】

```
/************************************
        代入演算子
************************************/
#include <stdio.h>

main()
{
        int     x=5;
        int     y=2;
        int     z1,z2;
        double  z3,z4;

        z1 = x / y;             /*(int)   /(int)->(int)*/
        z2 = (double)x / y;     /*(double)/(int)->(int)*/
        z3 = x/y;               /*(int)   /(int)->(double)*/
        z4 = (double)x / y;     /*(double)/(int)->(double)*/

        printf("z1 = %d ¥n",z1);  /* ① */
        printf("z2 = %d ¥n",z2);  /* ② */
        printf("z3 = %f ¥n",z3);  /* ③ */
```

【実行結果】

```
z1 = 2              …①
z2 = 2              …②
z3 = 2.000000       …③
z3 = 2.500000       …④
```

① int型/int型の結果をint型に代入
② double型/int型の結果をint型に代入。代入の段階で小数点以下が切り捨てられる
③ int型/int型の結果をdouble型に代入。除算の段階で切り捨てられてしまった情報は、もはや、元には戻らない
④ double型/int型の結果を double型に代入

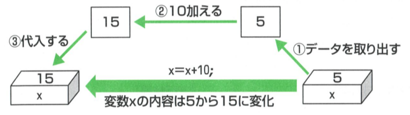
次ページに続く

```
    printf("z3 = %f ¥n",z4);  /* ④ */
}
```

●x = x + 10;

x = x + 10;という式は、数学ではありえません。が、C言語では、= は代入ですから、この式は、変数xからデータを取り出して、10加えた結果をもう一度変数xに戻すことになります。

▼x = x + 10;の演算例：xの初期値が5の場合



●複合代入演算子

x = x + 10;のような式をコンパクトに書けるように、C言語では、算術演算子と代入演算子を組み合わせた複合代入演算子が用意されています。

▼複合代入演算子

複合代入演算子	意味	一般形	同義の式
+=	加算の結果を代入	E1 += E2	E1 = E1 + E2
-=	減算の結果を代入	E1 -= E2	E1 = E1 - E2
*=	乗算の結果を代入	E1 *= E2	E1 = E1 * E2
/=	除算の結果を代入	E1 /= E2	E1 = E1 / E2
%=	剰余を代入	E1 %= E2	E1 = E1 % E2

複合代入演算子は、+と=の2つで1つの記号ですから、=+のように逆に書くとコンパイルエラーとなります。

4-11 増分・減分演算子と配列処理

配列に記録されているデータを1つずつ処理していくときなど、添え字の値を1ずつ増やしたり、1ずつ減らしたりすることがよくあります。複合代入演算子の加算を「1加える」に限定した演算子を増分演算子、減算を「1減らす」に限定した演算子を減分演算子といいます。

●添え字と増分・減分演算子

配列に記録されているデータを順に表示するには、表示したいデータの数だけprintf文を書きます。

【プログラム例】

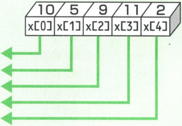
```
/************************************
        配列データの表示
************************************/
#include <stdio.h>

main()
{
        /*配列の初期化と宣言*/
        int     x[5] = {10,5,9,11,2};

        /*配列の表示*/
        printf("x[%1d] = %3d ¥n",0,x[0]);
        printf("x[%1d] = %3d ¥n",1,x[1]);
        printf("x[%1d] = %3d ¥n",2,x[2]);
        printf("x[%1d] = %3d ¥n",3,x[3]);
        printf("x[%1d] = %3d ¥n",4,x[4]);
}
```

【実行結果】

```
x[0] =  10
x[1] =   5
x[2] =   9
x[3] =  11
x[4] =   2
```



添え字の部分に変数を使ってみると、プログラムは、次のように書けます。

【書き換えたプログラム例】

```
/************************************
      配列データの表示
************************************/
#include <stdio.h>

main()
{
        /*配列の初期化と宣言*/
        int     x[5] = {10,5,9,11,2};
        int     i;                /*添え字*/

        /*配列の表示*/
        i = 0;
        printf("x[%1d] = %3d ¥n",i,x[i]);      i=0
        i++;
        printf("x[%1d] = %3d ¥n",i,x[i]);      i=1
        i++;
        printf("x[%1d] = %3d ¥n",i,x[i]);      i=2
        i++;
        printf("x[%1d] = %3d ¥n",i,x[i]);      i=3
        i++;
        printf("x[%1d] = %3d ¥n",i,x[i]);      i=4
        i++;      i=5
}
```

添え字を変数にし、増分演算子を使って1つずつ増やしていくことで、

```
printf("x[%1d] = %3d ¥n",i,x[i]);
i++;
```

というまったく同じ形の式が並ぶことになりました。かえって面倒になった、と思われる方もあるかもしれませんが、まったく同じ式が並ぶということは、後でとても重要になってきます。きっと、いいことがあります！

最後の**i++**（これで**i**は5になります）は、本来はなくてもよい余計なものですが、あっても害がないので、まったく同じ形の2式のペアを作るために、つけ加えてあります。

●前置増分・前置減分と後置増分・後置減分

増分・減分演算子には、演算子が式の前にあるもの（前置）と後ろにあるもの（後置）とがあります。前ページの例のように単項で使う場合にはどちらも同じですが、この演算子がほかの演算子を含む式の中で使われるときは、両者に違いがあります。

▼演算の種類の表し方

演算の種類	一般形	同義の式	式の値
前置増分	++E	E = E+1	増分後の値
後置増分	E++	E = E+1	増分前の値
前置減分	--E	E = E-1	減分後の値
後置減分	E--	E = E-1	減分前の値

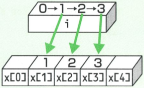
前置演算子は、先にEが増分（減分）され、変化したあとのEの値が使われます。iを先に増分してから代入されるので、x[0]には入りません。

【プログラム例】

```
/************************************
        前置増分
************************************/
#include <stdio.h>

main()
{
        /*配列の初期化と宣言*/
        int     x[5];
        int     i;        /*添え字*/

        /*前置演算子*/
        i = 0;
        x[++i] = i;
        x[++i] = i;
        x[++i] = i;
}
```



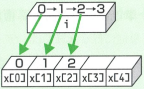
後置演算子は、増分（減分）される前のEの値が使われ、あとでEが変化します。増分する前のiを使って代入を行い、最後にiは1加えられますから、x[3]には入りません。

【プログラム例】

```
/************************************
        後置増分
************************************/
#include <stdio.h>

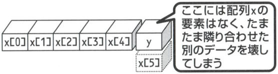
main()
{
        /*配列の初期化と宣言*/
        int     x[5];
        int     i;          /*添え字*/

        /*前置演算子*/
        i = 0;
        x[i++] = i;
        x[i++] = i;
        x[i++] = i;
}
```



●添え字の有効範囲

配列の要素数が5つのとき、添え字は0～4の範囲でなければなりません。増分演算子の使い方を誤って、添え字の有効範囲を超えてしまうことになると、隣接する異なる意味を持つ変数や配列領域に書き込んでしまうことになります。Cコンパイラは、添え字の妥当性を判定することができません。それは、プログラムを書く「あなた」の責任なのです。



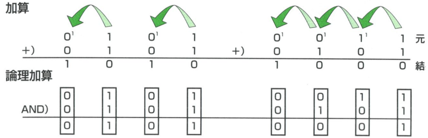
4-12 ビット単位の演算子

加算や減算などの算術演算子は、データ全体で演算を行います。下のケタからの繰り上がりや、上のケタからの繰り下がりがあります。これに対し、他のケタの演算結果の影響をまったく受けず、各ビットごとの演算を行う演算子をビット単位の演算子といいます。

●ビット単位に行う演算と算術演算のちがい

ビット単位に行う演算では、データの中の1ビットを取り出して、演算を行います。隣のビットからは影響を受けません。

▼ 各ビットごとに演算を行い、隣のビットに影響を与えない



●真理値表

ある特定のビットを取り出すと、そのデータは、'0'か'1'かのどちらかです。ですから、ビット単位の演算子では、演算子の右側のデータと左側のデータの1ビットの組み合わせは、

(0, 0) (0, 1) (1, 0) (1, 1)

の4通りしかありません。これらのすべての組み合わせについて演算した結果がどうなるかを表にしたものを、**真理値表**といいます。

●ビット単位の演算子の種類

ビット単位の演算子の種類は以下のとおりです。

▼ビット単位の演算子の種類

ビット単位の演算子	演算の種類	一般形	備考
&	ビット単位のAND	E1 & E2	真理値表参照
|	ビット単位のOR	E1 | E2	
^	ビット単位の排他的OR	E1 ^ E2	
~	ビット単位の補数	~E1	
<<	ビット単位の左シフト	E1 << E2	E1のビットをE2回左シフト
>>	ビット単位の右シフト	E1 >> E2	E1のビットをE2回右シフト

▼ビット単位の演算子の真理値表

E1	E2	E1 & E2	E1 | E2	E1 ^ E2	~E1
0	0	0	0	0	
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	

●ビットのセット・リセット・反転

ビット単位のAND演算子を使うと、データの中の指定のビットを0にする（リセットする）ことができます。

	0	1	0	0	1	1	0	1	元のデータ	(0x4d)
AND)	1	1	1	1	1	0	1	1		(0xfd)
	0	1	0	0	1	0	0	1	結果	(0x49)

他のビットはそのままで、このビットだけを、リセットします。

ビット単位のOR演算子を使うと、データの中の指定のビットを1にする（セットする）ことができます。

	0	1	0	0	1	1	0	1	元のデータ	(0x4d)
OR)	0	0	0	0	0	0	1	0		(0x02)
	0	1	0	0	1	1	1	1	結果	(0x4f)

他のビットはそのままで、このビットだけを、セットします。

ビット単位の排他的OR演算子を使うと、データの中の指定のビットを反転することができます。

	0	1	0	0	1	1	0	1	元のデータ	(0x4d)
XOR)	0	0	0	1	0	1	0	0		(0x14)
	0	1	0	1	1	0	0	1	結果	(0x59)

他のビットはそのままで、このビットだけを、反転します。

【プログラム例】

```
#include <stdio.h>

main()
{
        int     data=0x4d;

        printf("元のデータは  %2.2x¥n",data);
        printf("ビットリセット%2.2x¥n",data & 0xfb);
        printf("ビットセット  %2.2x¥n",data | 0x02);
        printf("ビット反転    %2.2x¥n",data ^ 0x14);
}
```

【実行結果】

```
元のデータは    4d
ビットリセット  49
ビットセット    4f
ビット反転      59
```

※上の解説図参照

●ビットのシフトは2倍・半分

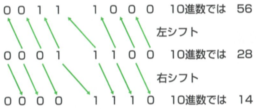
演算子<<は、ビットを1つずつ左にずらします。最上位ビットは捨てられ、最下位ビットには0が入ります。反対に、>>は、ビットを右にずらし、最上位ビットに0が入り、最下位ビットが捨てられます。<<は、2で掛けた結果と、>>は2で割った結果と同じになります。

これはたとえば、10進数で表現したときに、桁を右に1つずらすと1/10され、

左に1つずらすと10倍したことになるのと同じです。

```
          10倍              1/10
1230       ←       123       →       12.3
```

ビットの表現は2進数ですから、右へずらせば半分、左へずらせば2倍ということになります。



【プログラム例】

```
#include <stdio.h>

main()
{
        int     data = 0x1c;
        int     i;

        printf("右3回シフト  %2.2x --- (10進数)%3d¥n",data >> 3,data >> 3);
        printf("右2回シフト  %2.2x --- (10進数)%3d¥n",data >> 2,data >> 2);
        printf("右1回シフト  %2.2x --- (10進数)%3d¥n",data >> 1,data >> 1);
        printf("元のデータは %2.2x --- (10進数)%3d¥n",data,data);
        printf("左1回シフト  %2.2x --- (10進数)%3d¥n",data << 1,data << 1);
        printf("左2回シフト  %2.2x --- (10進数)%3d¥n",data << 2,data << 2);
        printf("左3回シフト  %2.2x --- (10進数)%3d¥n",data << 3,data << 3);
}
```

【実行結果】

```
右3回シフト  03 --- (10進数)   3
右2回シフト  07 --- (10進数)   7
右1回シフト  0e --- (10進数)  14
元のデータは 1c --- (10進数)  28
左1回シフト  38 --- (10進数)  56
左2回シフト  70 --- (10進数) 112
左3回シフト  e0 --- (10進数) 224
```

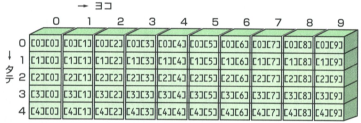
4-13 2次元配列を使って表を作ろう

情報を整理するときは、表を使うと便利なことがたくさんあります。表は、縦と横それぞれの項目に該当する欄にデータが入りますから、表の形式でデータを保持するには、縦横2つの添え字が必要です。このような配列を2次元配列といいます。

● 2次元配列とは

配列は、同じ型のデータをひとまとめに扱ったものでしたが、さらにこれを積み上げてひとまとめに扱ったものを**2次元配列**といいます。2次元配列は添え字を2つ使って表します。1次元配列が長屋であるのに対し、2次元配列は高層マンションのようです。

▼ 2次元配列の高層マンション



● 2次元配列の宣言

2次元配列を宣言するときは、縦の要素数と横の要素数とをそれぞれ指定します。配列全体の要素数は、縦の要素数×横の要素数となります。

2次元配列の宣言は、

```
型指定子　配列名[縦の要素数][横の要素数]={初期化子の並び};
```

の形式で行います。

●2次元配列のメモリの割り当てと初期化子の順序

2次元配列を宣言すると、そのすべての要素は、連続したアドレスに割り当てられることが保証されます。このとき、下の例のように行（横）が優先して連続し、初期化子も、この順に各要素に入ります。

【プログラム例】

```
/***********************************
        2次元配列の宣言
***********************************/
main()
{
        /*配列の宣言*/
        int     data[2][3] ={10,10,10,21,22,23};
}
```

data	0	1	2
0	10	10	10
1	21	22	23

data[0][0]	10
data[0][1]	10
data[0][2]	10
data[1][0]	21
data[1][1]	22
data[1][2]	23

記憶の順

このような場合、下の例のように、初期化子を列ごとに{ }でくくって書くと、列の区切りが明確になります。できるだけ、下の例のように書くように心がけてください。

【プログラム例】

```
        /*配列の宣言*/
        int     data[2][3] = {
                        {10,10,10}
                        {21,22,23}
        };
```

●2次元配列を使って九九の表を作ってみよう

1×1から9×9までの掛け算の表を作ってみましょう。縦・横それぞれ9つずつ要素をもつ2次元配列を宣言し、あらかじめ掛け算の結果を初期化します。これを行ごとに表示すると、九九表になります。

【プログラム例】

```
/*****************************************
        九九の表
*****************************************/
#include <stdio.h>

main()
{
        int     m[9][9]={
                        {1,2,3,4,5,6,7,8,9},                    /*1の段*/
                        {2,4,6,8,10,12,14,16,18},               /*2の段*/
                        {3,6,9,12,15,18,21,24,27},              /*3の段*/
                        {4,8,12,16,20,24,28,32,36},             /*4の段*/
                        {5,10,15,20,25,30,35,40,45},            /*5の段*/
                        {6,12,18,24,30,36,42,48,54},            /*6の段*/
                        {7,14,21,28,35,42,49,56,63},            /*7の段*/
                        {8,16,24,32,40,48,56,64,72},            /*8の段*/
                        {9,18,27,36,45,54,63,72,81}             /*9の段*/
        };

printf("     %4d %4d %4d %4d %4d %4d %4d %4d %4d¥n",1,2,3,4,5,6,7,8,9);
printf("--------------------------------------------------¥n");
printf("%4d¦%4d %4d %4d %4d %4d %4d %4d %4d %4d¥n",1,m[0][0],m[0][1],
m[0][2],m[0][3],m[0][4],m[0][5],m[0][6],m[0][7],m[0][8]);
printf("%4d¦%4d %4d %4d %4d %4d %4d %4d %4d %4d¥n",2,m[1][0],
m[1][1],m[1][2],m[1][3],m[1][4],m[1][5],m[1][6],m[1][7],m[1][8]);
printf("%4d¦%4d %4d %4d %4d %4d %4d %4d %4d %4d¥n",3,m[2][0],m[2][1],
m[2][2],m[2][3],m[2][4],m[2][5],m[2][6],m[2][7],m[2][8]);
printf("%4d¦%4d %4d %4d %4d %4d %4d %4d %4d %4d¥n",4,m[3][0],m[3][1],
m[3][2],m[3][3],m[3][4],m[3][5],m[3][6],m[3][7],m[3][8]);
printf("%4d¦%4d %4d %4d %4d %4d %4d %4d %4d %4d¥n",5,m[4][0],m[4][1],
m[4][2],m[4][3],m[4][4],m[4][5],m[4][6],m[4][7],m[4][8]);
printf("%4d¦%4d %4d %4d %4d %4d %4d %4d %4d %4d¥n",6,m[5][0],m[5][1],
```

次ページに続く

```
m[5][2],m[5][3],m[5][4],m[5][5],m[5][6],m[5][7],m[5][8]);
printf("%4d¦%4d %4d %4d %4d %4d %4d %4d %4d %4d¥n",7,m[6][0],m[6][1],
m[6][2],m[6][3],m[6][4],m[6][5],m[6][6],m[6][7],m[6][8]);
printf("%4d¦%4d %4d %4d %4d %4d %4d %4d %4d %4d¥n",8,m[7][0],m[7][1],
m[7][2],m[7][3],m[7][4],m[7][5],m[7][6],m[7][7],m[7][8]);
printf("%4d¦%4d %4d %4d %4d %4d %4d %4d %4d %4d¥n",9,m[8][0],m[8][1],
m[8][2],m[8][3],m[8][4],m[8][5],m[8][6],m[8][7],m[8][8]);

}
```

【実行結果】

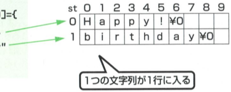
```
        1   2   3   4   5   6   7   8   9
------------------------------------------------
   1|   1   2   3   4   5   6   7   8   9
   2|   2   4   6   8  10  12  14  16  18
   3|   3   6   9  12  15  18  21  24  27
   4|   4   8  12  16  20  24  28  32  36
   5|   5  10  15  20  25  30  35  40  45
   6|   6  12  18  24  30  36  42  48  54
   7|   7  14  21  28  35  42  49  56  63
   8|   8  16  24  32  40  48  56  64  72
   9|   9  18  27  36  45  54  63  72  81
```

●2次元配列と文字列

1つの文字列は、char型の配列を使いますので、複数の文字列を扱うには2次元配列が必要です。行ごとに1つの文字列を記録します。

【プログラム例】

```
char  st[2][10]={
      "Happy!",
      "birthday"
};
```



4-14 アドレスを扱うポインタ変数

これまでは、データとして、数値や文字を扱ってきました。また、変数や配列はすべてメモリにロードされ、そのアドレスと密接な関係にあることも見てきました。C言語では、数値や文字とまったく同じように、アドレスをデータとして扱うことができます。アドレス型の変数をポインタといいます。

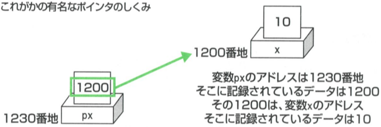
●アドレスと変数の関係

「変数を1つ確保し、ここにデータを代入する」ことを考えてみましょう。int型変数xが、たとえば1200番地にロードされたとすると、

```
int     x;
x = 10;
```

は、「1200番地から始まるint型の領域の内容を10とする」ということです。10はデータですが、C言語では、アドレス1200もデータとして、「10」とまったく同様に扱うことができます。xとは別の変数pxを確保すると、変数pxもどこかのアドレスにロードされます。そのアドレスを1230番地としましょう。そして、そこにデータとしての1200番地を代入したとすると、2つの変数の関係は下図のようになります。

▼これがかの有名なポインタのしくみ



このようにアドレスを保持する変数を**ポインタ**といいます。

●ポインタの宣言

C言語では、すべてのデータはいくつかの型に分類されます。型によって、必要なサイズ（バイト数）やデータの表現方法が異なることはすでに学習しました。ポインタ自身は、アドレスを記憶する変数ですから、アドレスを表現するのに必要なバイト数が確保されます。しかし、問題はそれだけではありません。ポインタが持つデータはアドレスですから、そのアドレスには、いったい何型のデータが入るのか、という情報が必要なのです。



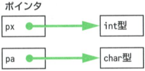
そこで、ポインタを宣言するときには、これから宣言しようとする変数がポインタであることを明らかにすると同時に、このポインタの指す先の型も明らかにしなければなりません。ポインタ型変数の宣言は、

```
型指定子　*変数名　;
```

の形式で行います。ここで、型指定子は、ポインタが指す先の型を示し、変数名の前につけた「*」で自分がポインタであることを表します。

【プログラム例】

```
int     *px;
char    *pa;
```



4-15 ポインタ専用の演算子

C言語には、変数や配列に割り当てられているアドレスを知る演算子が用意されています。また、ポインタに記録されているデータはアドレスですから、そのアドレスのデータを読み出したり、そこにデータを書き込んだりする演算子もあります。加減算は数値に対する演算とは異なります。

●アドレスを求めるアドレス演算子

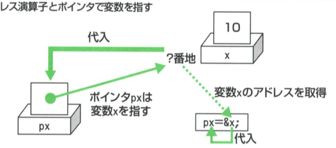
同じプログラムを何台かのマシンで実行したとき、プログラム内で使われている変数や配列が割り当てられるアドレスは、マシンごとに異なります。同じマシンであっても、その時々のマシンの状態によって、実行のたびにアドレスは異なるのです。ですから、変数のアドレスをポインタに記録するときには、そのときに割り当てられているアドレスを取得しなければなりません。それを行う演算子を**アドレス演算子**といいます。

▼アドレス演算子

アドレス演算子	演算の種類
&	アドレスを取得する

アドレス演算子を使ってある変数のアドレスを取得したら、これをポインタに代入すると、このポインタは、その変数を「指す」ことになります。

▼アドレス演算子とポインタで変数を指す



ためしに、アドレスを取得して表示してみましょう。

【プログラム例】

```
/*******************************************
        アドレス演算子
*******************************************/
#include <stdio.h>

main()
{
        int     x = 10;           /*int型の変数*/
        char    moji = 'a';       /*char型の変数*/
        int     *px;              /*int型を指すポインタ*/
        char    *pa;              /*char型を指すポインタ*/

        /*ポインタの設定*/
        px = &x;                         /* pxはxを指すようにする*/
        pa = &moji;                      /* paはmojiを指すようにする*/

        /*結果の表示*/
        printf("pxのアドレスは %p¥n",&px);
        printf("pxの内容は     %p¥n",px);
        printf(" xのアドレスは %p¥n",&x);
        printf(" xの内容は     %d¥n¥n",x);

        printf("  paのアドレスは %p¥n",&pa);
        printf("  paの内容は     %p¥n",pa);
        printf("mojiのアドレスは %p¥n",&moji);
        printf("mojiの内容は     %c¥n",moji);
}
```

%pは、アドレスを表示するための変換指定子

【実行結果】

```
pxのアドレスは     0064FDF8
pxの内容は         0064FE00  } 同じ
 xのアドレスは     0064FE00
 xの内容は         10
paのアドレスは     0064FDF4
paの内容は         0064FDFF  } 同じ
mojiのアドレスは   0064FDFF
mojiの内容は          a
```

アドレスが16進数で表記されています。この数値はシステムによって異なります。両者が等しいことを確認してください

●ポインタが指す先のデータを得る間接演算子

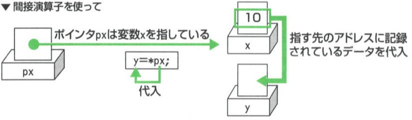
ポインタ変数は、数値や文字をデータに持つ普通の変数を指すことができますが、指しただけでは何の存在価値もありません。指す先のデータを"遠隔操作"することが重要なのです。遠隔操作とは、ポインタが指す先のアドレスに記録されているデータを読み出したり、ポインタの指す先のアドレスにデータを書き込んだりすることです。「ポインタの指す先のアドレスの内容」を表す演算子を**間接演算子**といいます。

▼間接演算子

間接演算子	演算の種類
*	ポインタの指す先

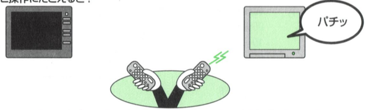
間接演算子の記号*は、ポインタを宣言したときに使った「*」と同じ記号です。ポインタ変数の宣言のときは、「この変数はポインタです」という意味になるのですが、この同じ記号が式中で使われているときは「指す先の内容」という異なった意味になります。両者を混同しないよう、ご注意ください。

▼間接演算子を使って



ちょうど、リモコンで遠くから、テレビを操作するのと似ています。

▼テレビ操作にたとえると？



リモコンを右のテレビに向けると、右のテレビを操作できます。

【プログラム例】

```
/******************************************
        間接演算子
******************************************/
#include <stdio.h>

main()
{
        int     x = 10;         /*int型の変数*/
        int     *px;            /*int型を指すポインタ*/

        /*ポインタの設定*/
        px = &x;          /* pxはxを指すようにする*/

        /*ポインタを使って変数の内容を読み出す*/
        printf("pxの指す先の内容は  %d¥n",*px);

        /*ポインタを使って変数の内容を書き換える*/
        *px = 20;

        printf("¥n  pxの指す先の内容を書き換えました¥n");
        printf("pxの指す先の内容は  %d¥n",*px);
}
```

pxは、int型を指す「私はポインタ」です

pxの指す先の内容を表示

pxの指す先のアドレスに20を代入する

pxの指す先の内容を表示

【実行結果】

```
pxの指す先の内容は  10

  pxの指す先の内容を書き換えました
pxの指す先の内容は  20
```

C言語の=は代入ですから、左辺に演算子を書くことはできません。が、間接演算子だけは、例外的に左辺に書くことができます。

4-16 配列を指すポインタ

ポインタは、アドレスを直接扱いますから、その指す先が変数であろうと配列であろうと、ポインタは、まったく気にしません。同じように間接演算子で指す先の内容を遠隔操作できます。が、アドレスを得るときは配列と変数とで異なります。配列名はアドレスを持ちますので、アドレス演算子が不要となります。

●配列名とアドレス

1つの変数には1つのデータしか入りませんので、式の右辺に変数名を書くと、この変数の内容、つまり中に記録されているデータが処理の対象となります。これに対して配列名は、「この配列の先頭の要素のアドレス」という意味になります。ですから、配列の先頭のアドレスをポインタに設定するときには、アドレス演算子を書かず、直接配列名を代入します。

【プログラム例】

```
/*****************************************
        配列を指すポインタ
*****************************************/
#include <stdio.h>

main()
{
        int     x = 10;                         /*変数の宣言*/
        int     xx[5] = {20,21,22,23,24};       /*配列の宣言*/
        int     *px,*pxx;                       /*ポインタの宣言*/

        /*ポインタにアドレスを設定*/
        px = &x;
        pxx = xx;

        /*表示*/
```

（注釈）
- int *px,*pxx; → px pxxはint型を指す「私はポインタ」
- px = &x; → 変数xのアドレスを代入
- pxx = xx; → 配列xxのアドレスを代入

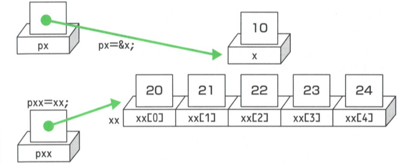
【実行結果】

```
pxの指す先の内容は10
pxxの指す先の内容は20
```

次ページに続く

```
        printf(" pxの指す先の内容は%d¥n",*px);
        printf("pxxの指す先の内容は%d¥n",*pxx);
}
```

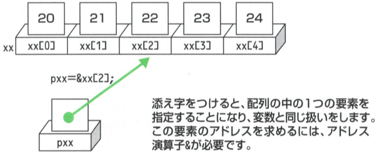
▼プログラム例の図解



●ポインタが配列の途中の要素を指す

ポインタは、配列の途中の要素を指すこともできます。このときは、配列の1つの要素を変数のように扱って、添え字とアドレス演算子を使って、指したい要素のアドレスを得ます。

▼配列の途中の要素を指すポインタ



4-17 ポインタの加減算

ポインタの内容はアドレスですが、アドレスも数値ですから、整数値を加えたり、2つのアドレスの差をとったりすることができます。が、通常の数値の加減算とは異なり、指す先の型に応じた加減算となります。ポインタが配列を指すとき、ポインタの加算は添え字と同じ意味を持ちます。

●配列の添え字とポインタ

配列では、添え字が1増えると、その次の要素を表します。

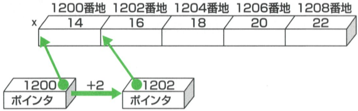


添え字の値が1増えると、となりの要素を表す

P.115のプログラム例を思い出してください。添え字を1ずつ足して、次々と要素の内容を表示しました。

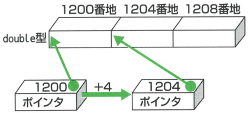
ポインタの内容はアドレスですから、次の要素を指すためには、ポインタが指す先の型のサイズ分を加えなければなりません。たとえば、ポインタがint型の配列を指している場合、int型が2バイトであれば、ポインタの値に2を加えたアドレスが次の要素になります。

▼ int型（2バイト）の配列を指すポインタが、次の要素を指す場合



ところが、ポインタが指す先がdouble型の配列であり、double型が4バイトであれば、次の要素を指すためには、アドレスに4を加えなければなりません。

▼double型（4バイト）の配列を指すポインタが、次の要素を指す場合

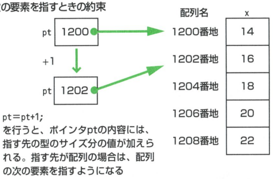


このように、ポインタが指す先の型によって、加える値が異なるのは、とても面倒です。それを解消するため、ポインタに1を加えることで、そのポインタが指す先の型のサイズ分が自動的に加えられるように設計されました。ポインタは、その指す先の型が何であろうとも、

```
pt = pt + 1;
```
（ptポインタ変数とする）

と書くと、次の要素を指すことができます。

▼ポインタが次の要素を指すときの約束



【プログラム例】

```
/******************************************
          ポインタの加減算
******************************************/
#include <stdio.h>

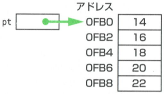
main()
{
        int     x[5]={14,16,18,20,22};   /*配列の宣言*/
        int     *px;                     /*ポインタの宣言*/

        /*ポインタの設定*/
        px = x;

        /*データの表示*/
        printf("%p番地の内容は %3d ¥n",px,*px);
        px++;
        printf("%p番地の内容は %3d ¥n",px,*px);
        px++;
        printf("%p番地の内容は %3d ¥n",px,*px);
        px++;
        printf("%p番地の内容は %3d ¥n",px,*px);
        px++;
        printf("%p番地の内容は %3d ¥n",px,*px);
        px++;
}
```

【実行結果】

```
OFB0番地の内容は  14
OFB2番地の内容は  16
OFB4番地の内容は  18
OFB6番地の内容は  20
OFB8番地の内容は  22
```



px++の演算の結果、
アドレスが2ずつ増えています

（注意）この数値はアドレスなので、システムによって異なります。2ずつ増えていることに注目してください。

●加減算と間接演算子

前ページのプログラムではポインタの内容を次々に変えて表示を行いましたが、

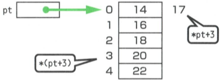
*(px+1)

のように書くと、「ポインタpxの内容に1加えた、具体的には、指す先の型のバイト数分加えたアドレスの内容」という意味になります。ここで、間接演算子*と加算+とでは、*の方が演算の優先順が上なので、

*px+1

は、「pxの指す先の内容に1を加える」という意味になってしまいます。これは、ちょうど、式中に加算と乗算が混じっているときは、必ず乗算を先に行う、という算数の規則に似ています。

5 ＋ 3 × 2 ＝ 5 ＋ 6 ＝ 11
こちらを先に行う



【プログラム例】

```
/**************************************
        ポインタの加減算と間接演算子
**************************************/
#include <stdio.h>

main()
{
        int     x[5]={14,16,18,20,22};    /*配列の宣言*/
        int     *px;                       /*ポインタの宣言*/

        /*ポインタの設定*/
        px = x;

        /*データの表示*/
        printf("*(px+3)：pxから3番目の指す先は        %3d ¥n",*(px+3));
        printf("*px+3  ：pxの指す先の内容に3加える %3d ¥n",*px+3);
}
```

【実行結果】

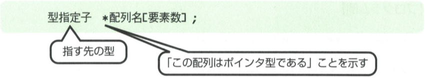
```
*(px+3)：pxから3番目の指す先は        20
*px+3  ：pxの指す先の内容に3加える    17
```

4-18 ポインタ配列

ポインタをたくさん並べて配列にしたものをポインタ配列といいます。各要素は、それ自身がすべてポインタでなければならないことはもちろんですが、指す先の型も、皆同じでなければなりません。ポインタ配列を使うと、ばらばらのデータを一元的に管理することができます。

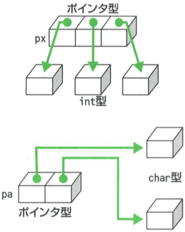
●ポインタ配列の宣言

ポインタ型であっても、配列の宣言は、int型やchar型のそれとまったく同様です。その変数がポインタ型であること、要素数、そして、ポインタの場合は特別に、指す先の型を明らかにします。



【プログラム例】

```
/********************************
ポインタ配列の宣言
********************************/
main()
{
        int    *px[3];
        char   *pa[2];
}
```



●配列要素の指す先は

ポインタ配列は、各要素のそれぞれが、決められた型の変数や配列を指します。各要素は、添え字を使って指定し、その指す先を表すには、間接演算子*を使います。

【プログラム例】

```
/************************************
        ポインタ変数の指す先
************************************/
#include <stdio.h>

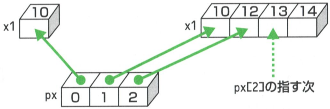
main()
{
        int     *px[3];

        int     x1 = 10;
        int     x2[4] = {11,12,13,14};
        /*ポインタをつなぐ*/
        px[0] = &x1;
        px[1] = x2;
        px[2] = &(x2[1]);

        printf("px[0]の指す先は：%4d¥n",*px[0]);
        printf("px[1]の指す先は：%4d¥n",*px[1]);
        printf("px[2]の指す先は：%4d¥n",*px[2]);
        printf("px[2]の指す次は：%4d¥n",*(px[2]+1));
        printf("px[0]の次の要素の指す先は：%4d¥n",*px[0+1]);
}
```

【実行結果】

```
px[0]の指す先は：   10
px[1]の指す先は：   11
px[2]の指す先は：   12
px[2]の指す次は：   13
px[0]の次の要素の指す先は：   11
```



4-19 何も指さないポインタNULL

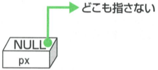
ポインタは、いつもどこか有効なメモリを指しているわけではありません。まだ指す先のアドレスが決まらないこともありますし、指していた先のデータが無効になってしまうこともあります。そのように、ポインタがどこも指していないときには、ポインタの値は特別なものにします。

●NULLポインタ

ポインタがどこも指していないとき、慣例的に、ポインタの値は0にします。メモリには、0番地は存在するのですが、アプリケーションのデータが0番地に入ることは絶対にありません。そこで、「ポインタの値が0のときは、どこも指していないことにする」という約束をしておきます。このように、ポインタの値としての0は特別な意味を持ちますので、これにNULL（ヌル）という名前がつけられました。

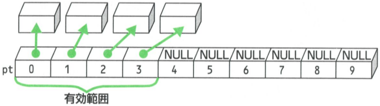
【プログラム例】

```
int  *px = NULL;
```



●ポインタ配列の有効範囲を示すNULL

有効なデータの数が状況によって変化するプログラムでは、ポインタ配列の要素数を多めに用意しておきます。この要素のうち、先頭から必要な分の要素だけが有効なデータを指していて、残りはどこも指していないNULLとしておきます。

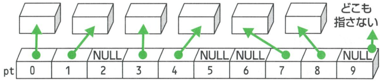


このようなシステムでは、「ポインタ配列の配列要素がいくつあるか」ということよりも、「どこまでが有効か」ということの方が重要なのです。図でpt[4]は、有効なデータの終わりを示す役割を担います。

●有効なデータが飛び飛びのときのNULL

有効なデータが飛び飛びにある例では、ポインタ配列の全部の要素1つ1つについて、その有効性を確認してから処理を行う必要があります。そのためには、有効でない要素にはNULLを入れておかなければなりません。ポインタがNULLでないときだけ、指す先のデータに処理を行うようにするのです。つまり、各要素にとって、その値がNULLかどうかはとても重要なことなのです。

割り算で、0で割ってはならないのと同様に、ポインタでは、NULLの指す先に値を代入したり、参照したりといった操作をしてはならないのです。



One Point 用語解説

➡定数名

プログラムの中で、数字を使うことはよくあります。たとえば、配列の要素数を指定するときなどです。また、データとして特別な意味を持たせることもあります。たとえば、性別を記録するchar型変数seibetuに、男だったら、'M'、女だったら'W'を代入するとしましょう。プログラムでは、

```
char seibetu = 'M';
```

のように書くことができますが、これでは、データ'M'の意味がはっきりしません。

そこで、データ'M'にMEN、データ'W'にWOMENという名前をつけ、'M'や'W'のかわりにこの名前が使えたら、初めてプログラムを見た人でも変数seibetuの意味やそこに記録されたデータの意味を容易に理解できることでしょう。このような決まったデータ、つまり定数につけた名前を定数名といいます。定数名は#defineを使って定義します。

【例】
```
#define MEN   'M'
#define WOMEN 'W'

char seibetu = MEN;
```

定数名は、変数名や配列名と区別するために慣例的に大文字にします。NULLは、ポインタ値ゼロを表す定数名です。

4-20 キーボードからのデータの入力

プログラムの中でデータを代入したり、初期化したりしていたのでは、データが変わるたびにコンパイルしなおさなければなりません。キーボードからデータを入力することができれば、その場でデータに合わせた処理を行うことができます。

●書式つき入力 scanfで整数を入力

キーボードから整数を入力するには、データを受け取るint型の変数が必要です。また、入力されたデータが「整数であるとみなす」という意味の記号"%d"を使います。この記号は、printf文のときと同じです。データを受け取る変数は、そのアドレスをscanfに渡します。

【scanf文の一般形】

```
scanf("書式", 変数のアドレス);
```

【プログラム例】

```
/*************************************
        int型データの入力
*************************************/
#include <stdio.h>

main()
{
        int     x;

        /*入力*/
        printf("int型データを入力:");
        scanf("%d",&x);

        printf("入力されたデータは%dです¥n",x);
}
```

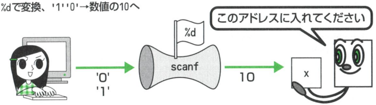
【実行結果】

```
int型データを入力: 10
入力されたデータは10です
```

■ の部分はキーボードから入力

キーボードから入力された情報は、初めは、すべて文字です。"10"は数値の10ではなく、'1'と'0'なのです。%dは、それを数値の10に変換する働きをしてくれます。また、入力されたデータをどこにしまうのか、そのアドレスが必要になりますので、アドレス演算子&を使います。

▼ %dで変換、'1''0'→数値の10へ



●浮動小数点数の入力

浮動小数点数を入力する場合の書式は、受け取る変数がdouble型であれば"%lf"、float型であれば"%f"となります。

【プログラム例】

```
/****************************************
         double型データの入力
****************************************/
#include <stdio.h>

main()
{
         double  xd;
         float   xf;

         /*入力*/
         printf("double型データを入力:");
         scanf("%lf",&xd);
         printf("float型データを入力:");
         scanf("%f",&xf);

         printf("xdに入力されたデータは%6.1fです¥n",xd);
         printf("xfに入力されたデータは%6.1fです¥n",xf);
}
```

【実行結果】

```
double型データを入力: 10.5
float型データを入力: 10.2
xdに入力されたデータは    10.5です
xfに入力されたデータは    10.2です
```

□ の部分はキーボードから入力

●文字の入力

1文字を入力する場合の書式は、"%c"となります。

【プログラム例】

```
/****************************************
        char型データの入力
****************************************/
#include <stdio.h>

main()
{
        char    a;

        /*入力*/
        printf("char型データを入力:");
        scanf("%c",&a);

        printf("変数aに入力されたデータは %c です¥n",a);
}
```

【実行結果】

```
char型データを入力: A
変数aに入力されたデータは A です
```

▭ の部分はキーボードから入力

●文字列の入力

文字列を入力する場合の書式は、"%s"です。文字列の場合は、入力されるデータが複数文字ですから、入力データを受け取るためには、char型の配列が必要になります。scnafには、格納したい先頭の要素のアドレスを渡します。配列名は、先頭のアドレスを持ちますから、配列の先頭から入力データを格納する場合には、アドレス演算子&は不要です。scnafは、指定のアドレスから順に入力データを格納します。配列要素がいくつ用意され、どこまで使っていいのか、ということはまったく知らずに、入力されたデータを全部次々に格納しますから、十分な数の配列要素を用意する必要があります。

入力文字列の最後には、自動的に文字列の終わりを表す'¥0'がつきます。

【プログラム例】

```
/*************************************
        文字列データの入力
*************************************/
#include <stdio.h>

main()
{
        char    a[20];

        /*入力*/
        printf("文字列を入力:");
        scanf("%s",a);
        printf("文字列を入力:");
        scanf("%s",&a[10]);

        printf("1つ目の文字列は %s です¥n",a);
        printf("2つ目の文字列は %s です¥n",&a[10]);
}
```

【実行結果】

```
文字列を入力: Happy
文字列を入力: birthday!
1つ目の文字列は Happy です
2つ目の文字列は birtuday! です
```

□の部分はキーボードから入力



One Point 用語解説

➡ **プロンプト**

どのプログラムも、scanfの直前に、printf文を書いて、どのようなデータを入力するのか、が表示されるようにしてあります。このprintf文がないと、画面では、予告なくカーソルが点滅した、入力待ち状態となってしまうのです。

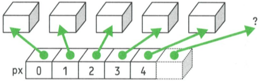
これでは、ユーザは、マシンがロックしたのではないか、と錯覚してしまいます。このprintf文は、ユーザからの入力待ちのために停止していることをアピールし、入力を促す役割があります。つまり、このプログラムの中でのプロンプトとなっているのです。MS-DOSのプロンプトと同様です。プログラム開発には、そのプログラムを使うユーザに対して、入力操作にとまどうことのないよう、このくらいの気遣いが欲しいものです。

COLUMN

低頻度バグ

ある操作を行うと必ず起こる不具合は、原因の究明が比較的簡単です。その操作をやってみて、そのときの、メモリの状態やプログラムの実行の状況をチェックすれば、大方は解決します。しかし、たまにしか発生しない不具合は、とてもやっかいです。同じ操作をしても、うまくいくときと、いかないときがある…。そんなときは、原因のとらえどころがありません。このようなバグを低頻度バグといいます。

ポインタ配列pxを使ってデータを参照する例を考えてみましょう。配列要素数が5個のときは、添え字は0から4までのどれかでなければなりませんね。



ここで、まちがってpx[5]の指す先に何かを書き込んだとします。本来、配列pxはpx[4]までしかありませんが、配列はアドレスが連続している、という原則にしたがって、次の領域をpx[5]だと思ってしまいます。が、ここには、別の変数や配列が割り当てられていて、まったくちがうデータが入っているのです。それは、ポインタ型ではなく、int型やchar型かもしれません。それをアドレスとみなして、その先に何かを書き込むのですから…。どこに書くことになるのか、予想がつきません。こうして低頻度バグは発生してしまうのです。

COLUMN

5章

プログラムの流れを制御してみよう

コンピュータは、同じことを何度も繰り返すのが得意です。また、言われたとおり忠実に働きますから、複雑な条件も1つ1つ確実にこなしてくれます。この章では、そんなコンピュータの特性を生かして、状況に対応したプログラムを書いていきます。3つの基本構造の考え方を実際にプログラムしてみましょう。

5-1 条件を満たす・満たさない

「もし、○○だったら」というときの○○を、C言語では「条件」といいます。「条件を満たす間、△△を繰り返す」や「条件を満たした場合のみ□□をする」というように、条件は反復構造や選択構造には欠かせないものです。

●論理値

今、午前中ですか？　と聞かれたら、「はい」か「いいえ」かで答えます。「午前11時です。」と答えるのは、もしかしたら親切かもしれませんが、そんなことは聞かれていませんね。このように、「はい」と「いいえ」の2種類しか答え（値）のないものを**論理値**といいます。C言語には、論理値を専用に扱う型はなく、int型で代用します。

▼論理値を表すint型

条件を	論理値		int型の値
満たす	true	真	1（非零）
満たさない	false	偽	0

●関係演算子

「条件」の多くは、あるデータが○○と比べてどうか、という大小関係です。文字の場合の大小関係は、その文字のコードを比べることで得られます。このような演算子を**関係演算子**といい、結果は0（真）・1（偽）のいずれかとなります。

▼関係演算子

関係演算子	演算の種類	備考
<	小さい	
>	大きい	
<=	以下	2つで1つの記号　=< は不可
>=	以上	2つで1つの記号　=> は不可
==	等しい	= では、右辺から左辺に代入される
!=	等しくない	

【プログラム例】

```
/***************************************
	関係演算子
***************************************/
#include <stdio.h>

main()
{
	int	x;

	/*入力*/
	printf("int型の値を入力：");
	scanf("%d",&x);

	/*結果*/
	printf("xは%3d  x>3 は  %d¥n",x,x>3);
}
```

【実行結果】

```
int型の値を入力：10
xは 10  x>3 は  1
```

真のとき、結果は1

の部分はキーボードから入力

```
int型の値を入力：1
xは 1  x>3 は  0
```

偽のとき、結果は0

●論理演算子

「○○である　かつ　△△である」などという条件を複合条件といい、それを求める演算子を**論理演算子**といいます。論理演算子は、int型どうしで演算しますが、その値を論理値とみなします。

▼論理演算子

論理演算子	演算の種類	備考
&&	～かつ～	演算の結果は真理値表による
||	～または～	
!	～でない	

▼真理値表

E1	E2	E1 && E2	E1 || E2	!E1
真（非零）	真（非零）	真（非零）	真（非零）	
真（非零）	偽（0）	偽（0）	真（非零）	偽（0）
偽（0）	真（非零）	偽（0）	真（非零）	真（非零）
偽（0）	偽（0）	偽（0）	偽（0）	

5-2 条件による処理の選択 ～選択構造①～

条件により、どちらか片方の処理だけが実行される構造を選択構造といいます。プログラム中に、2通りの処理が書かれていても、そのどちらか一方しか実行されないのです。2つの処理を選択するための条件として、論理値が得られる式を書きます。

●if文による選択

2つのうちのどちらかを選択するには、if文を使います。

if文の一般形は、

```
if（論理式）
{
        処理1：論理式が真（true）のときに行う処理
}
else
{
        処理2：論理式が偽（false）のときに行う処理
}
```

処理1と処理2のどちらか片方しか実行されません。また、論理式が偽のときに行う処理がないときは、else以下を省略することができます。が、真のときに行う処理がない場合には、何も省略できません。そのようなときには、条件判定を反対にします。

```
if（気温が30℃以下だったら）
{
        何もしない
}
else
{
        冷房を入れる
}
```

→

```
if（気温が30℃を超えたら）
{
        冷房を入れる
}
```

【プログラム例】

キーボードから性別を入力して、これを表示します。入力は、男なら0、女なら1とし、表示は　男　または　女　とします。

```
#include <stdio.h>

main()
{
        int     seibetu;        /*性別*/

        /*データの入力*/
        printf("性別は？  ");
        scanf("%d",&seibetu);

        /*表示*/
        printf("性別：");
        if (seibetu == 0)
        {
                printf("男");
        }
        else
        {
                printf("女");
        }
        printf("¥n");
}
```

- if (seibetu == 0)：ここには;は書かない
- printf("男");：0が入力されたら　男
- printf("女");：そうでなければ　女

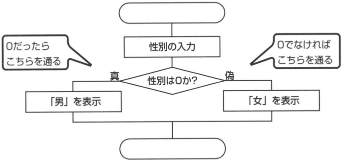
【実行結果】

```
性別は？  0
性別：男
```

```
性別は？  1
性別：女
```

（網かけ）の部分はキーボードからの入力

【流れ図】



5-3 多分岐からの選択 ～選択構造②～

選択肢は、いつも2つの中から1つを選ぶだけとは限りません。たくさんの選択肢の中から、たった1つを選ぶ多肢選択であることもあります。そのようなときには、else if文を用い、○○ではないとき、△△か？　などのように、次々と条件を書きます。

●else if文による多肢選択

選択肢が3つ以上ある場合には、else if文を使います。

else if文の一般形は、

```
if（論理式1）
{
        処理1：論理式1が真（true）のときに行う処理
}
else if （論理式2）
{
        処理2：論理式1が偽（false）で、論理式2
               が真（true）のときに行う処理
}
else
{
        処理3：論理式1も論理式2も、偽（false）
               のときに行う処理
}
```

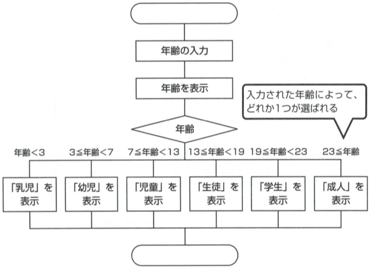
処理1と処理2と処理3は、どれか1つしか実行されません。処理3に何もすることがない場合には、else以下の文は省略することができますが、その他は省略できません。else if文は、いくつでも書くことができます。

【プログラムの内容】

キーボードから年齢を入力して、以下の表にしたがって表示します。

年齢	表示	年齢	表示
0～2才	乳児	13～18才	生徒
3～6才	幼児	19～22才	学生
7～12才	児童	23才以上	成人

【流れ図】



【実行結果】

```
年齢は？ 10
年齢：10才　児童
```

□ の部分はキーボードからの入力

```
年齢は？ 13
年齢：13才　生徒
```

```
年齢は？ 30
年齢：30才　成人
```

【プログラム例】

```
#include <stdio.h>

main()
{
        int     nenrei;

        /*入力*/
        printf("年齢は？");
        scanf("%d",&nenrei);

        /*表示*/
        printf("年齢：%d才　",nenrei);
        if(nenrei<3)
        {
                printf("乳児");      3才未満だったら「乳児」
        }
        else if(nenrei<7)
        {
                printf("幼児");      そうでなくて7才未満だったら「幼児」
        }
        else if(nenrei<13)
        {
                printf("児童");      そうでなくて13才未満だったら「児童」
        }
        else if(nenrei<19)
        {
                printf("生徒");      そうでなくて19才未満だったら「生徒」
        }
        else if(nenrei<23)
        {
                printf("学生");      そうでなくて23才未満だったら「学生」
        }
        else
        {
                printf("成人");      そうでなければ「成人」
        }
}
```

●switch文による多肢選択

条件となるデータが整数または文字で、その取りうる値が限られているときには、switch文を使うと便利です。

【switch文の一般形】

```
switch (式)
{
    case 定数式1:
        文1;
        break;
    case 定数式2:
        文2;
        break;
         :
         :
    case 定数式n:
        文n:
        break;
    default:
        文n+1;
}
```

式の値が定数式1に等しいときには文1が、定数式2に等しいときには文2が、…定数式nに等しいときには文nが、該当なしのときには、文n+1が選ばれます。該当なしのとき、何もする処理がなければ、defaultとそれに続く文は省略することができます。各条件の各文の最後にbreak;がありますが、これを省略すると、次の条件の文を続けて実行します。

One Point 用語解説

➡文とブロック

C言語では、1つの命令を文といいます。文の最後には、必ず、「;」(セミコロン)を付けます。変数を宣言する

```
int    x;
```

も文ですし、画面に表示する

```
printf("私の名前は太郎です");
```

も文です。これらの文を複数集めて{ }で囲ったものをブロックといい、処理のまとまりを表します。

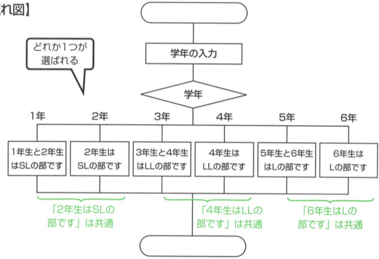
if文の、真のときに行う処理・偽のときに行う処理などは、それぞれブロックです。また、プログラム全体を囲む{ }もブロックです。

【プログラムの内容】

キーボードから学年を入力し、これにより、以下のように表示します。

学年	表示
6年	6年生は　L　の部です
5年	5年生と6年生は　L　の部です
4年	4年生は　LL　の部です
3年	3年生と4年生は　LL　の部です
2年	2年生は　SL　の部です
1年	1年生と2年生は　SL　の部です
その他	学年が間違っています

【流れ図】



【実行結果】

学年を入力してください：6
6年生は　L　の部です

学年を入力してください：5
5年生と6年生は　L　の部です

　　の部分はキーボードからの入力

【プログラム例】

```
#include <stdio.h>

main()
{
        int           gakunen;

        printf("学年を入力してください：");
        scanf("%d",&gakunen);

        switch(gakunen){
                case    5:      /*5年生*/
                        printf("5年生と");
```

break;がないので、次を続けます

```
                case    6:      /*6年生*/
                        printf("6年生は　L　の部です¥n");
                        break;
```

break;があるので、ここまでしか実行しません

```
                case    3:      /*3年生*/
                        printf("3年生と");
```

break;がないので、次を続けます

```
                case    4:      /*4年生*/
                        printf("4年生は　LL　の部です¥n");
                        break;
```

break;があるので、ここまでしか実行しません

```
                case    1:      /*1年生*/
                        printf("1年生と");
```

break;がないので、次を続けます

```
                case    2:      /*2年生*/
                        printf("2年生は　SL　の部です¥n");
                        break;
```

break;があるので、ここまでしか実行しません

```
                default:                /*その他*/
                        printf("学年がまちがっています¥n");
        }
}
```

5-4 同じ文の繰り返し ～反復構造①～

データを変えて、同じ処理を何度も行う、ということはよくあります。このとき、同じ処理を何度も書くのではなく、1回書いた処理を繰り返すことができれば、プログラムは柔軟なものになります。このようなプログラムの構造を反復構造といいます。

●繰り返し処理と回数のカウント

画面に同じ文字を3回表示することにしましょう。

```
「FC希望ヶ丘ファイト」　「オー！」
「FC希望ヶ丘ファイト」　「オー！」
「FC希望ヶ丘ファイト」　「オー！」
```

プログラムはどうなるでしょうか?

【プログラム例】

```
#include <stdio.h>

main()
{
        printf("「FC希望ヶ丘ファイト」　");
        printf("「オー！」¥n");

        printf("「FC希望ヶ丘ファイト」　");
        printf("「オー！」¥n");

        printf("「FC希望ヶ丘ファイト」　");
        printf("「オー！」¥n");
}
```

上のプログラムのように、まったく同じ処理を3回書けば、3回表示すること

はできます。白枠内の部分を1回だけ書いて、これを「3回繰り返す」ようにできれば、もっと柔軟なプログラムになります。なぜなら、繰り返す回数さえ変えれば、自由に表示回数を変えることができるからです。

さて、「3回繰り返す」ためには、今、何回繰り返したか、を数えていなければなりません。その役目は、int型変数「i」に担ってもらいましょう。

▼ 変数iで、繰り返しを数える



初めは、1回も表示していませんので、変数iは0にしておきます。

```
printf("「FC希望ヶ丘ファイト」  ");
printf("「オー!」¥n");
```



1回表示したので、変数iを1にします。

```
printf("「FC希望ヶ丘ファイト」  ");
printf("「オー!」¥n");
```



2回表示したので、変数iを2にします。

```
printf("「FC希望ヶ丘ファイト」  ");
printf("「オー!」¥n");
```



3回表示したので、変数iを3にします。
3回表示したので、終わりです。

初めにiを0にすることを初期化といいます。iは、1回表示するごとに、1ずつ大きくしていきますから、i++が使えます。この処理を再初期化といいます。「iが3になったら終わり」は「iが3より小さい間、繰り返す」と言い換えることができます。これを反復条件といいます。

●while文による繰り返し

繰り返しを行うには、while文を使います。

【while文の一般形】

```
while (論理式)
{
        繰り返しを行う文;
}
```

論理式の値が真である間、{ }内の処理を繰り返します。前ページのプログラムをwhile文を使って書き換えると、以下のように書けます。

【プログラム例】

```
#include <stdio.h>

main()
{
        int     i;        /*繰り返しを数える変数*/

①──→   i = 0;            /*初期化*/
        while(i<3)        /*iが3より小さい間、繰り返す*/
        {
                /*繰り返す処理*/
                printf("「FC希望ヶ丘ファイト」  ");
                printf("「オー！」¥n");

                i++;      /*再初期化*/
        }
}
```

①の(i<3)を(i<5)にしさえすれば、同じ処理を5回繰り返すことができます。キーボードから入力した回数だけ繰り返して表示してみるように修正したプロ

グラムは、以下のようになります。

【プログラム例】

```
#include <stdio.h>

main()
{
        int     n;      /*繰り返す回数*/
        int     i;      /*繰り返しを数える変数*/

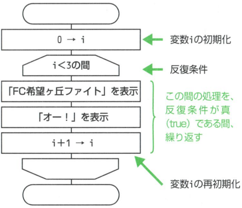
        /*回数の入力*/
        printf("何回表示しますか？");
        scanf("%d",&n);

        i = 0;          /*初期化*/
        while(i<n)      /*iがnより小さい間、繰り返す*/
        {
                /*繰り返す処理*/
                printf("「FC希望ヶ丘ファイト」   ");
                printf("「オー！」¥n");

                i++;    /*再初期化*/
        }
}
```

繰り返し回数がnに入る

【流れ図】



5-5 for文による反復表現 ～反復構造②～

while文を使った反復構造では、初期化・反復条件・再初期化を書く位置が、それぞれバラバラであるために、ついうっかり忘れてしまうこともあります。for文は、これらを統合して書く方法です。統合することで、コンパクトで忘れものの少ないプログラムにすることができます。

●for文による繰り返し

for文は、while文とまったく同じ機能を持っています。

【for文の一般形】

```
for(初期化 ; 論理式 ; 再初期化)
{

    繰り返しを行う文;

}
```

for文は、以下の順に実行されます。

① 初期化（1回のみ）
② 論理式の判定→真であれば
③ 繰り返し文
④ 再初期化
⑤ 論理式の判定→真であれば
⑥ 繰り返し文
⑦ 再初期化

:

論理式の判定→偽となれば、反復の終了

前節のプログラムをfor文を使って書き換えると、以下のように書けます。

【プログラム例】

```
#include <stdio.h>

main()
{
        int     n;          /*繰り返す回数*/
        int     i;          /*繰り返しを数える変数*/

        /*回数の入力*/
        printf("何回表示しますか？");
        scanf("%d",&n);

        for(i=0;i<n;i++)          /*iがnより小さい間、繰り返す*/
        {
                /*繰り返す処理*/
                printf("「FC希望ヶ丘ファイト」　");
                printf("「オー！」¥n");
        }
}
```

●繰り返さない例

while文もfor文も、繰り返し処理を行う前に、反復条件の判定をしますから、条件によっては、1回も反復処理を行わないで反復を終了する可能性があります。たとえば上のプログラムで、「何回表示しますか?」の問いに対し、0が入力された場合には、反復条件i<nは、

　　iが0、nが0

となって初めから偽となり、繰り返す処理

```
printf("「FC希望ヶ丘ファイト」　");
printf("「オー！」¥n");
```

は1度も実行されません。

5-6 配列のデータと繰り返す処理～反復構造③～

同じ処理の繰り返しとはいえ、まったく同じことを繰り返すことはまれです。多くは、異なったデータに対して、同じ処理を施します。このような場合には、データを配列に入れておくと、反復構造を用いることができます。文を同じ形にするための工夫が必要です。

●異なるデータに対する繰り返し

下のように表示することを考えます。

```
「FC希望ヶ丘ファイト」   「オー！」
「霧が丘FC　ファイト」   「オー！」
「星が丘SS　ファイト」   「オー！」
「夢ヶ丘SC　ファイト」   「オー！」
```

表示する文字が1つ1つ異なり、このままでは、反復構造が使えません。

【プログラム例】

```
#include <stdio.h>

main()
{
        printf("「FC希望ヶ丘 ファイト」    ");
        printf("「オー！」¥n");

        printf("「霧が丘FC   ファイト」    ");
        printf("「オー！」¥n");

        printf("「星が丘SS   ファイト」    ");
        printf("「オー！」¥n");

        printf("「夢ヶ丘SC   ファイト」    ");
        printf("「オー！」¥n");

}
```

白窓部が1つ1つ異なるため、反復構造にできない

そこで、表示したい文字列を配列に入れておきます。1つの文字列を記録するには、char型の1次元配列が必要です。日本語は、char型配列の要素2個分で1文字を表します。このような文字を2バイト文字といいます。文字列としてひとくくりに扱うと、2バイト文字も通常の半角の文字と同様に扱えます。

ここでは、文字列が4つありますから、2次元配列を宣言し、各行に1つの文字列を入れます。

```
char team[4][14];          /*二次元配列の宣言*/
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
team[0]	F	C	希		望		ヶ		丘		¥0			
team[1]	霧		が		丘		F	C			¥0			
team[2]	星		が		丘		S	S			¥0			
team[3]	夢		ヶ		丘		S	C			¥0			

この配列を使ってプログラムを書き換えると以下のようになります。

【書き換えたプログラム例】

```
#include <stdio.h>

main()
{
        char    team[4][14]={"FC希望ヶ丘","霧が丘FC   ",
                             "星が丘SS   ","夢ヶ丘SC   "};

        printf("「%sファイト」    ",team[0]);
        printf("「オー！」¥n");

        printf("「%sファイト」    ",team[1]);
        printf("「オー！」¥n");

        printf("「%sファイト」    ",team[2]);
        printf("「オー！」¥n");

        printf("「%sファイト」    ",team[3]);
        printf("「オー！」¥n");

}
```

文字列を直接書くかわりに、変換指定子を書く。実際に表示する内容として、配列を指定する

添え字が異なる以外は、2行ずつ、4つの処理がまったく同じ形になった

反復構造を使うためには、添え字さえも、全部同じになっていなければなりません。そこで、変数**i**に注目してみましょう。

▼繰り返しを数える変数iを、添え字に使う

0 i

変数iの値は0なので、0の代わりにiと書けます。

```
printf("「%sファイト」 ",team[0]);
printf("「オー!」¥n");
```

1 i

1回表示したので、変数iを1にします。

```
i++;
```

```
printf("「%sファイト」 ",team[1]);
printf("「オー!」¥n");
```

2 i

2回表示したので、変数iを2にします。

```
i++;
```

```
printf("「%sファイト」 ",team[2]);
printf("「オー!」¥n");
```

3 i

3回表示したので、変数iを3にします。

```
i++;
```

```
printf("「%sファイト」 ",team[3]);
printf("「オー!」¥n");
```

4 i

4回表示したので、変数iを4にします。

```
i++;
```

4回表示したので、終わりです。

添え字に変数iを使うと、4つの処理がすべて同じになりました。これで反復構造が使えます。

for文を使ったプログラムは以下のようになります。

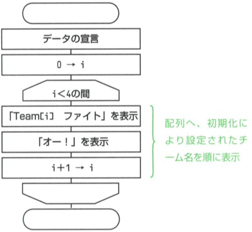
【プログラム例】

```
#include <stdio.h>

main()
{
        char    team[4][14]={"FC希望ヶ丘","霧が丘FC    ","星が丘SS    ",
                              "夢ヶ丘SC    "};

        int     i;                    /*添え字*/
                                              4回繰り返す
        for(i=0;i<4;i++)
        {
                printf(" 「%sファイト」    ",team[i]);
                printf(" 「オー！」¥n");
        }
}
```

【流れ図】

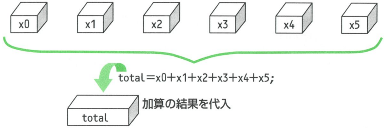


5-7 合計を算出するアルゴリズム

配列に入っている複数の数値データの合計値を求めます。合計するデータが確定している場合には、加算のら列でも合計を求めることができますが、ここでは、合計するデータの数が可変の場合にも対応できるアルゴリズムを考えます。

●配列のデータの加算

いくつかのデータがあるとき、それらの合計を求める最も簡単な方法は、加算を並べて書く方法です。

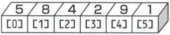


ところが今度は、x2～x4までの合計を求めなければならなくなったときには、文を書き換えなければなりません。

```
total = x2 + x3 + x4;
```

柔軟なプログラムを書くために、まず、データは配列に入れます。

```
int     x[6]={5,8,4,2,9,1};
```



次に合計を求める変数を0で初期化します。ここに、それぞれのデータを1つずつ足していきます。

【プログラム例①】

```
#include <stdio.h>

main()
{
        int     x[6]={5,8,4,2,9,1};        /*データ*/
        int     total;                     /*合計*/

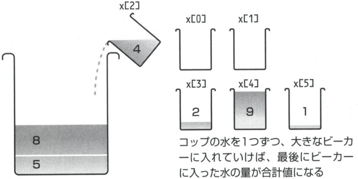
        total = 0;

        total = total + x[0];
        total = total + x[1];
        total = total + x[2];
        total = total + x[3];
        total = total + x[4];
        total = total + x[5];

        /*結果の表示*/
        printf("x[0]からx[5]までの合計は%d¥n",total);
}
```

似たような文が6つ並びました。違うのは添え字の数値だけです。今回も、変数iを使ってまったく同じ文になるように書き換えます。

▼コップの水を合計する例えで考えると…



コップの水を1つずつ、大きなビーカーに入れていけば、最後にビーカーに入った水の量が合計値になる

【プログラム例②】

```
#include <stdio.h>

main()
{
        int     x[6]={5,8,4,2,9,1};         /*データ*/
        int     total;                      /*合計*/
        int     i;

        total = 0;                          /*合計の初期化*/
        i=0;
        total = total + x[i];
        i++;
        total = total + x[i];
        i++;
        total = total + x[i];   まったく同じ2文が6回並んだ
        i++;
        total = total + x[i];
        i++;
        total = total + x[i];
        i++;
        total = total + x[i];
        i++;

        /*結果の表示*/
        printf("x[0]からx[5]までの合計は%d¥n",total);
}
```

(注意) このプログラムをコンパイルすると、使用したコンパイラによっては、最後のi++;の文に対し、「変数iに代入された値は使われていない」という趣旨の警告が表示されることがありますが、形を同じにするために挿入した文で、その後このiは使われないことは承知していますので、この警告は無視してかまいません。

●○番目から△番目までの合計

上のプログラムでは、6つまったく同じ形の文ができましたので、これで反復構造が使えます。反復構造では、変数iで順に添え字を指し示しながら加算を進めていきますから、配列の何番目のデータから何番目のデータまでを加算

するのか、を自由に指定できます。始めと終わりをキーボードから入力して、その範囲だけを加算するプログラムを書いてみましょう。

【プログラム例】

```
#include <stdio.h>

main()
{
        int     x[6]={5,8,4,2,9,1};         /*データ*/
        int     total;                      /*合計*/
        int     i;                          /*添え字*/
        int     hajime;                     /*○番目から*/
        int     owari;                      /*△番目まで*/

        printf("何番目から？  ");
        scanf("%d",&hajime);
        printf("何番目まで？  ");
        scanf("%d",&owari);

        total = 0;                          /*合計の初期化*/

        for(i=hajime; i<=owari; i++)
        {
                total = total + x[i];
        }

        /*結果の表示*/
        printf("x[%d]からx[%d]までの合計は%d¥n",hajime,owari,total);
}
```

hajimeからowariまで、順に加算

【実行結果】

```
何番目から？  1
何番目まで？  4
x[1]からx[4]までの合計は23
```

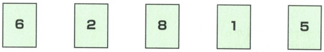
の部分はキーボードからの入力

5-8 最大値・最小値を求めるアルゴリズム

配列に入っている複数の数値データの中から、最大値と最小値を求めます。データ同士を比べて大小関係を求めるためには、選択構造を使いますし、これをすべてのデータについて行うには、反復構造を使います。最大値・最小値は、2つの構造の組み合わせで求められます。

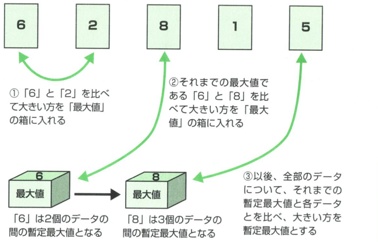
●最大値の求め方

下図のようなカードの中から最大値を求めます。



人間なら、しばらく見つめて、最大値は「8」。「左から3番目」ということを一瞬にして知ることができます。が、いったいどうやって求めているのでしょうか？

▼5枚の数字カードから、最大値のカードを選ぶ



これをこのままプログラムにしてみましょう。

【プログラム例】

```
#include <stdio.h>

main()
{
    int     x[5]={6,2,8,1,5};          /*データ*/
    int     max;                       /*最大値*/

    if(x[0]<x[1])
    {
        max = x[0];
    }
    else
    {
        max = x[1];
    }

    if(max < x[2])
    {
        max = x[2];
    }

    if(max < x[3])
    {
        max = x[3];
    }

    if(max < x[4])
    {
        max = x[4];
    }

    /*結果の表示*/
    printf("最大値は%d¥n",max);
}
```

ここだけ異なっている

初めに、先頭とその次を比べ、大きい方を暫定最大値にする

x[0]の方が大きいとき

x[1]の方が大きいとき

同じ文が並んでいる

暫定最大値よりx[2]の方が大きいとき、x[2]が暫定最大値になる

暫定最大値よりx[3]の方が大きいとき、x[3]が暫定最大値になる

暫定最大値よりx[4]の方が大きいとき、x[4]が暫定最大値になる

始めだけ、先頭とその次とを比べていますが、あとは、暫定最大値とデータとを比べています。全部同じ文にするために、初めに、maxをx[0]で初期化することにしましょう。この部分は、

```
max = x[0];

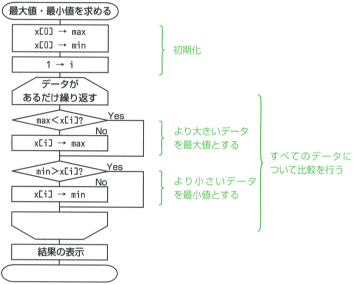
if(max<x[1])
{
        max = x[1];
}
```

となって、すべて同じになりました。あとは、添え字を変数iで置き換えれば、反復構造が使えます。

●最大値・最小値を求めるプログラム

最小値も、最大値とまったく同じ考え方で求められます。小さい方を暫定最小値にすればよいのです。

【流れ図】



【プログラム例】

```
#include <stdio.h>

main()
{
        int     x[5]={6,2,8,1,5};              /*データ*/
        int     max;                           /*最大値*/
        int     min;                           /*最小値*/
        int     i;                             /*添え字*/


        max = x[0];     /*最大値を最初のデータで初期化*/
        min = x[0];     /*最小値を最初のデータで初期化*/

        for(i=1;i<5;i++)
        {
                /*最大値を求める*/
                if(max < x[i])
                {                       /*暫定最大値より大きいとき*/
                        max = x[i];  /*このデータを暫定最大値とする*/
                }

                /*最小値を求める*/
                if(min > x[i])
                {                       /*暫定最小値より小さいとき*/
                        min = x[i];  /*このデータを暫定最小値とする*/
                }
        }

        /*結果の表示*/
        printf("最大値は%d¥n",max);
        printf("最小値は%d¥n",min);
}
```

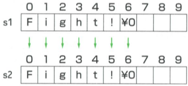
x[0]で初期化したので、iは1から始める

5-9 文字列を複写するアルゴリズム

char型配列に入っている文字列を別の配列に複写します。たとえ、文字列のようにひとまとめに扱うデータであったとしても、配列の複数の要素を一度に代入することはできません。各要素のデータを1つ1つ新しい配列の要素に代入します。この操作をポインタを使って実現します。

●文字列を複写する

char型配列に入っている文字列を別の配列に複写するには、下図のように要素の1つ1つを代入しなければなりません。



このままプログラムすると、

```
#include <stdio.h>

main()
{
        char    s1[10] = "Fight!";
        char    s2[10];

        s2[0] = s1[0];          /*'F'*/
        s2[1] = s1[1];          /*'i'*/
        s2[2] = s1[2];          /*'g'*/
        s2[3] = s1[3];          /*'h'*/
        s2[4] = s1[4];          /*'t'*/
        s2[5] = s1[5];          /*'!'*/
        s2[6] = s1[6];          /*'¥0'*/
```

次ページに続く

```
        printf("s1の文字列は%s¥n",s1);
        printf("s2の文字列は%s¥n",s2);
}
```

反復構造にするには、添え字を変数**i**で置き換えるのはこれまでと同じですが、問題は、「何回繰り返すか」という反復条件です。「添え字**i**が**6**になったら終わり」という条件では、長さの異なる文字列に対応できません。どんな文字列にも共通なのは「最後が**'¥0'**である」ということですから、「先頭から順に複写して、データが**'¥0'**となったら終わり」ということになります。これを言い換えると「**'¥0'**でない間、繰り返す」が反復条件になります。

【プログラム例】

```
#include <stdio.h>

main()
{
        char    s1[10] = "Fight!";          /*元の文字列*/
        char    s2[10];                     /*複写先*/
        int     i;                          /*添え字*/

        for(i=0;s1[i] != '¥0';i++)
        {
                s2[i] = s1[i];   /*複写*/
        }
        s2[i] = s1[i];

        printf("s1の文字列は%s¥n",s1);
        printf("s2の文字列は%s¥n",s2);
}
```

この条件では、s1[i]の内容が'¥0'になったら複写の処理を終えるため、このままでは、最後の'¥0'が複写されない

その分の複写を、最後に行う

●文字列を指すポインタ

上の処理を**char**型を指すポインタを使って実現します。ポインタに対する加算は、指す先の型のサイズ分を**1**として行われますから、上のプログラムの

i++;と、この配列を指すポインタpa++;とは、同じ意味になります。

【プログラム例】

```
#include <stdio.h>

main()
{
        char    s1[10] = "Fight!";          /*元の文字列*/
        char    s2[10];               /*複写先*/
        char    *ps1;                 /*s1を指すポインタ*/
        char    *ps2;                 /*s2を指すポインタ*/

        /*ポインタの設定*/
        ps1 = s1;
        ps2 = s2;

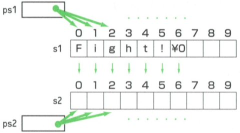
        /*複写*/
        for(;*ps1 != '¥0';ps1++,ps2++)
        {
                *ps2 = *ps1;      /*複写*/
        }
        *ps2 = *ps1;

        printf("s1の文字列は%s¥n",s1);
        printf("s2の文字列は%s¥n",s2);
}
```

ps1の指す先が'¥0'でない間繰り返す

初期化処理がないときは、;だけ書きます

上のプログラムは、ポインタの値をどんどん変えていきましたが、ポインタそのものの値は固定し、変数iを次々と増やしていく方法もあります。変数iは、配列の添え字と同じ働きをします。



【プログラム例】

```
#include <stdio.h>

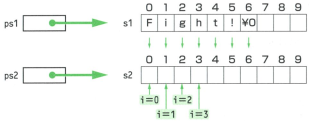
main()
{
        char    s1[10] = "Fight!";          /*元の文字列*/
        char    s2[10];             /*複写先*/
        char    *ps1;               /*s1を指すポインタ*/
        char    *ps2;               /*s2を指すポインタ*/
        int     i;                  /*添え字*/

        /*ポインタの設定*/
        ps1 = s1;
        ps2 = s2;

        /*複写*/
        for(i = 0;*(ps1+i) != '¥0';i++)
        {
                *(ps2+i) = *(ps1+i);        /*複写*/
        }
        *(ps2+i) = *(ps1+i);

        printf("s1の文字列は%s¥n",s1);
        printf("s2の文字列は%s¥n",s2);
}
```

ポインタの指す先から数えてi番目の要素を複写



One Point 用語解説

ソースとディスティネーション

データを複写するとき、複写元のデータをソース（source）、複写先の入れものをディスティネーション（destination）といいます。

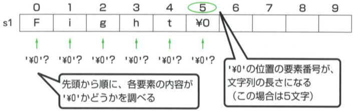
sourceとは、英語で「源」という意味です。一方、destinationは、英語で「到着地」「目的地」という意味があります。データの源から目的地へ、複写するのです。

5-10 文字列の長さを求めるアルゴリズム

char型配列に入っている文字列の長さ（文字数）を求めます。文字列は、必ず、その最後に'¥0'があります。文字列の先頭から順に1文字ずつ調べて、'¥0'のある位置が文字列の長さになります。この操作をポインタで実現します。

●'¥0'のある位置を調べる

char型配列に入っている文字列の長さ（文字数）を求めるには、下図のようにします。



このままプログラムすると、以下のようになります。

【プログラム例】

```
#include <stdio.h>

main()
{
        char    s1[9]="Fight";
        int     i;       /*添え字*/

        i = 0;
        if(s1[i] != '¥0')          /*'F'*/
        {
                i++;
```

配列要素の内容が'¥0'でないときは、添え字を1カウントアップし、次の要素を指すようにする

次ページに続く

```
        }
        if(s1[i] != '¥0')       /*'i'*/
        {
            i++;
        }
        if(s1[i] != '¥0')       /*'g'*/
        {
            i++;
        }
        if(s1[i] != '¥0')       /*'h'*/
        {
            i++;
        }
        if(s1[i] != '¥0')       /*'t'*/
        {
            i++;
        }
        if(s1[i] != '¥0')       /*'¥0'*/
        {
            i++;
        }

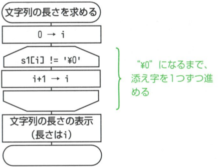
        printf("長さは%d文字です¥n",i);
}
```

配列要素の内容が'¥0'のときは、条件を満たさないのでi++されず、そのときのiが求める文字列の長さになる

●反復構造を使って

上のプログラムは、「配列の要素が'¥0'になるまで、配列の添え字iを次に進める」と言い換えることができます。さらに言い換えると「配列要素が'¥0'でない間、配列の添え字iを進める」ということになります。

【流れ図】



【プログラム例】

```
#include <stdio.h>

main()
{
        char    s1[9]="Fight";
        int     i;        /*添え字*/

        for(i = 0;s1[i] != '¥0';i++)
        {

        }

        printf("長さは%d文字です¥n",i);
}
```

i++をここに書くので、反復構造の本体には何も書くことがない

反復構造の部分は、空のブロックを書くかわりに、;を使って、以下のように書けます。

```
for(i = 0;s1[i] != '¥0';i++)
{

}
```

→

```
for(i = 0;s1[i] != '¥0';i++);
```

●ポインタを使って

配列を指すポインタを使って書き換えると、以下のようになります。複写のときと異なり、文字数を数えるための変数**i**が必要です。プログラム例①では、ポインタの値を次々に変えて**'¥0'**を探す方法を、プログラム例②では、変数**i**を使い、ポインタそのものの値は変化させずに**'¥0'**を探す方法を、それぞれ紹介します。

【プログラム例①】

```
#include <stdio.h>

main()
{
        char    s1[9]="Fight";
        char    *ps1;
        int     i;          /*添え字*/

        /*ポインタの設定*/
        ps1 = s1;

        for(i = 0;*ps1 != '¥0';ps1++,i++);

        printf("長さは%d文字です¥n",i);
}
```

変数iは、文字数を数える

【プログラム例②】

```
#include <stdio.h>

main()
{
        char    s1[9]="Fight";
        char    *ps1;
        int     i;          /*添え字*/

        /*ポインタの設定*/
        ps1 = s1;

        for(i = 0;*(ps1+i) != '¥0';i++);

        printf("長さは%d文字です¥n",i);
}
```

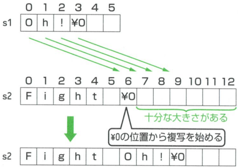
変数iは、添え字の働きをしている

5-11 文字列を合成するアルゴリズム

char型配列に入っている文字列の後ろへ別の文字列を合成するアルゴリズムを考えます。まず、前になる文字列はどこまでなのかを調べなければなりません。次に、その続きに、別の配列に入っている文字列の複写をしていきます。この操作をポインタで実現します。

●文字列を合成する

char型配列に入っている文字列の後ろに別の文字列を複写することで、2つの文字列を合成します。このとき、前になる文字列が入っている配列は、後ろに、別の文字列が入るだけの十分な大きさがあるものとします。文字列の合成は下図のように行います。



プログラムは、

① どこから複写を始めるかを探す

② 複写する

の2本立てになります。①の複写をする位置を探す処理は、前の文字列の'¥0'を探す処理になりますので、これは、文字列の長さを求めたときと同じです。②の複写する処理は、複写する先の要素番号（添え字）が、複写元のそれと異なることを除いては、文字列の複写と同じ処理になります。添え字が異なるの

ですから、複写先・複写元の2つの文字列の添え字に別々の変数を使えば解決できます。まず最初は、配列でプログラムを書いてみましょう。

【プログラム例】

```
#include <stdio.h>

main()
{
        char    s1[6] = "Oh!";
        char    s2[13] = "Fight ";
        int     i;      /*s1の添え字*/
        int     j;      /*s2の添え字*/

        for(j = 0;s2[j] != '¥0';j++);    ← '¥0'を探す

        for(i = 0;s1[i] != '¥0';i++,j++)  ┐
        {                                  │ 複写
                s2[j] = s1[i];             │
        }                                  ┘
        s2[j] = s1[i];    ← 最後の'¥0'の分

        printf("新しい文字列は %s¥n",s2);
}
```

【実行結果】

```
新しい文字列は Fight Oh!
```

●ポインタを使って

ポインタを使う場合には、今、ポインタがどこを指しているのかを、常に考えていなければなりません。そうでないと、ポインタは、データでないところを指し示し、そこに入っている違う意味のデータを使ってしまうことになります。

最初に、ポインタの値が次々に変化する方法で、次に、ポインタの値を変えずに変数を使う方法で、プログラムを書き換えてみましょう。

【プログラム例①】

```
#include <stdio.h>

main()
{
        char    s1[6] = "Oh!";
        char    s2[13] = "Fight ";
        char    *ps1;    /*s1を指すポインタ*/
        char    *ps2;    /*s2を指すポインタ*/

        /*ポインタの設定*/
        ps1 = s1;
        ps2 = s2;

        /*文字列の最後を探す*/
        for(;*ps2 != '¥0';ps2++);

        /*複写*/
        for(;*ps1 != '¥0';ps1++,ps2++)
        {
                *ps2 = *ps1;
        }
        *ps2 = *ps1;   最後の'¥0'の分

        printf("新しい文字列は %s¥n",s2);
}
```

One Point 用語解説

➡構造化プログラミング

どんなに複雑なアルゴリズムであっても、すべてのプログラムは3つの基本構造、

① 順次構造
② 選択構造
③ 反復構造

の組み合わせで書くことができる、と言われています。このような考え方を構造化プログラミングといいます。

以前は、メモリが高価であったり、コンピュータのハードウェアの性能が十分でなかったことなどから、コンパクトで高速なプログラムがよいプログラムと言われていましたが、既存のプログラムの修正に膨大な時間と経費がかかることから、現在では、修正しやすく、わかりやすいプログラムがよいとされています。構造化プログラミングは、各構造をブロック化することで、他に影響の少ないプログラムを書くことができるのです。

【プログラム例②】

```
#include <stdio.h>

main()
{
        char    s1[6] = "Oh!";
        char    s2[13] = "Fight ";
        char    *ps1;     /*s1を指すポインタ*/
        char    *ps2;     /*s2を指すポインタ*/
        int     i;        /*s1の添え字*/
        int     j;        /*s2の添え字*/

        /*ポインタの設定*/

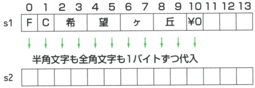
        ps1 = s1;
        ps2 = s2;

        /*文字列の最後を探す*/
        for(j=0;*(ps2+j) != '¥0';j++);

        for(i=0;*(ps1+i) != '¥0';i++,j++)
        {
                *(ps2+j) = *(ps1+i);
        }

        printf("新しい文字列は %s¥n",s2);
}
```

文字列のデータが、日本語（全角文字）であった場合でも、まったく同様に処理することができます。全角文字は2バイトで1文字を表しますが、複写するときは、1バイトずつバラバラに複写します。が、全部の複写が終われば、また2バイトずつ組にして扱えるので、問題はありません。



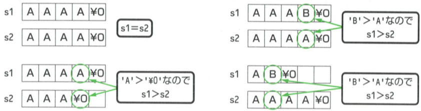
5-12 文字列の大小関係を求めるアルゴリズム

コンピュータ内部では、文字は、アスキーコード（数値）で表現されています。ですから、1文字の大小関係は、その文字に割り当てられたアスキーコードの大小関係となります。文字列の場合、大小関係は、辞書の順になります。辞書の中で、前にある文字列は、後ろの文字列より小さいことになります。

●辞書式配列

辞書順では、先頭の文字の小さい方が前になります。先頭の文字が同じときは、2番目の文字が小さい方が前になります。2番目も同じときは、3番目、3番目まで全部同じときは、4番目…と順に比べていきます。文字数と、全部の文字が等しいとき、2つの文字列は等しくなります。

例

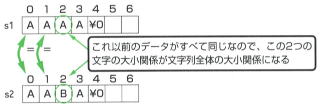


文字列の大小関係は、文字列の長さには関係がなく、それぞれの文字の持つコードによって決まります。

●辞書式配列のアルゴリズム

次ページ上のような図の例で考えます。

このままプログラムすると、右のようになります。



【プログラム例】

```
#include <stdio.h>

main()
{
        char    s1[7]="AAAA";
        char    s2[7]="AAB";
        int     i;                  /*添え字*/

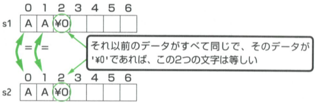
        i = 0;
        if(s1[i] == s2[i])          /*'A'*/
        {
                i++;
        }
        if(s1[i] == s2[i])          /*'A'*/
        {
                i++;
        }
        if(s1[i] == s2[i])          /*'A'と'B'*/
        {
                i++;
        }
        if(s1[i] > s2[i])
        {
                printf("s1 > s2");
        }
        else  if(s1[i] < s2[i])
        {
                printf("s1 < s2");
        }
}
```

s1[i]とs2[i]が等しいときは、次の要素を比べるためにiの値を1増やす

条件を満たさないので、i++しない

s1[i]とs2[i]の大きい方が、文字列全体の大きい方となる

もう1つ、次の例でも考えます。プログラムすると、以下のようになります。



【プログラム例】

```
#include <stdio.h>

main()
{
        char    s1[7]="AA";
        char    s2[7]="AA";
        int     i;                      /*添え字*/

        i = 0;
        if(s1[i] == s2[i] && s1[i] != '¥0')/*'A'*/
        {
                i++;
        }
        if(s1[i] == s2[i] && s1[i] != '¥0')/*'A'*/
        {
                i++;
        }
        if(s1[i] == s2[i] && s1[i] != '¥0')/*'¥0'*/
        {
                i++;
        }
        if(s1[i] > s2[i])
        {
                printf("s1 > s2");
        }
        else if(s1[i] < s2[i])
        {
                printf("s1 < s2");
        }
        else
        {
                printf("s1 = s2");
        }
}
```

s1[i]とs2[i]が'¥0'でなくて等しいときは、次の要素を比べるためにi++を行う

s1[i]は'¥0'であり、条件を満たさないので、i++;はしない

s1[i]とs2[i]が等しく、'¥0'のとき、2つの文字列は等しい

●反復構造を使って

これら2つのプログラムを総合すると、「2つの配列の要素**s1[i]**と**s2[i]**が**'¥0'**でなく等しい間、次の要素を指すように添え字**i**を**1**ずつ増す。条件を満たさなくなった**s1[i]**と**s2[i]**との大小関係が文字列全体の大小関係となる」ということになります。

比較する文字列をキーボードから入力する処理も追加して、反復構造で書き換えると、以下のようになります。

【プログラム例】

```
#include <stdio.h>

main()
{
        char    s1[10];
        char    s2[10];
        int     i;                    /*添え字*/

        /*文字列の入力*/
        printf("文字列1を入力：");
        scanf("%s",s1);
        printf("文字列2を入力：");
        scanf("%s",s2);

        for(i = 0;s1[i] == s2[i] && s1[i] != '¥0';i++);

        if(s1[i] > s2[i])
        {
                printf("文字列1 > 文字列2");
        }
        else if(s1[i] < s2[i])
        {
                printf("文字列1 < 文字列2");
        }
        else
        {
                printf("文字列1 = 文字列2");
        }
}
```

s1[i]とs2[i]が最初に等しくないか、または、'¥0'であるiを探す

i番目の要素の大小関係が文字列の大小関係になる

●ポインタを使って

ポインタの値を次々に変えていく方法と、ポインタの値は変えずに変数iを次々に変えていく方法の、2つのプログラム例を紹介します。

【プログラム例①】

```
#include <stdio.h>

main()
{
        char    s1[10];
        char    s2[10];
        char    *ps1,*ps2;          /*ポインタ*/

        /*ポインタの設定*/
        ps1 = s1;
        ps2 = s2;

        /*文字列の入力*/
        printf("文字列1を入力：");
        scanf("%s",ps1);
        printf("文字列2を入力：");
        scanf("%s",ps2);

        for(;*ps1 == *ps2 && *ps1 != '¥0';ps1++,ps2++);

        if(*ps1 > *ps2)
        {
                printf("文字列1 > 文字列2");
        }
        else if(*ps1 < *ps2)
        {
                printf("文字列1 < 文字列2");
        }
        else
        {
                printf("文字列1 = 文字列2");
        }
}
```

【プログラム例②】

```
#include <stdio.h>

main()
{
        char    s1[10];
        char    s2[10];
        char    *ps1,*ps2;          /*ポインタ*/
        int     i;                  /*添え字*/

        /*ポインタの設定*/
        ps1 = s1;
        ps2 = s2;

        /*文字列の入力*/
        printf("文字列1を入力：");
        scanf("%s",ps1);
        printf("文字列2を入力：");
        scanf("%s",ps2);

        for(i=0;*(ps1+i) == *(ps2+i) && *(ps1+i) != '¥0';i++);

        if(*(ps1+i) > *(ps2+i))
        {
                printf("文字列1 > 文字列2");
        }
        else if(*(ps1+i) < *(ps2+i))
        {
                printf("文字列1 < 文字列2");
        }
        else
        {
                printf("文字列1 = 文字列2");
        }
}
```

【実行結果】

```
文字列1を入力：AAAA
文字列2を入力：AAB
文字列1 < 文字列2
```

の部分はキーボードからの入力

5-13 処理のまとまりを関数にする

まとまった処理は、別の場所に書いて、必要なときに呼び出すことができます。これを関数といいます。汎用性の高い処理を関数にしておけば、一度だけ書いて何度も使うことができます。固有な処理を関数にすることで、全体を一瞥できる、見通しのよいプログラムにすることができます。

●関数とmain()の実行順

今までは、プログラムの中にはmain()のブロックしかなく、以下のような形式でした。

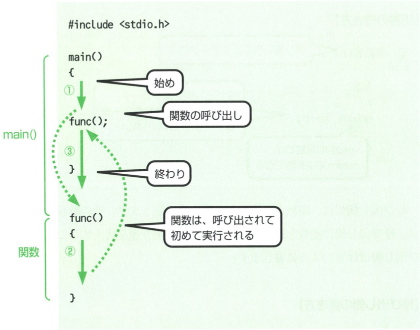
```
#include <stdio.h>

main()
{


}
```

次ページの図のように、main()とは別のブロックに、別のプログラムを書くことができます。これを識別するために名前をつけたものが**関数**です。

main()も関数の1つですが、「プログラムはここから始まる」という特別な役目を持っています。ですから、main()は、ブロックに必ず1つだけ存在しなければなりません。プログラムの実行が始まったら、main()のブロックに書かれた文が、原則として上から順に実行されます。このままでは、main()とは別ブロックに書かれた関数内の文は実行されません。main()など、実行中のブロックから呼び出されて、初めて関数内の文は実行されます。



上図で↓と①～③は、実行の順序です。関数の呼び出しがあると、main()の実行を中断して、関数の処理を実行します。関数の処理が終わると、main()の続きを実行します。同じ関数が何度も呼び出されると、呼び出された回数だけ実行されます。

●関数の書き方

関数は、処理した結果得られたデータを、呼び出し元に返すことができます。このとき、返すデータの型を「関数の型」といいます。int型のデータを返すときは、int型の関数であり、char型のデータを返すときは、char型の関数です。データを返さない関数は、void型の関数といいます。関数の型は、関数名の前に書いて、明らかにします。

呼び出し元に返すデータは、関数の最後に実行されるreturn文で指定します。関数部分は、次ページのように書きます。

【関数の書き方】

```
型　関数名()
{
        文;
        :
        return データ;
}
```

呼び出し側に返すデータの型

呼び出し側に返すデータ。データの型は関数の型と一致していなければならない

void型の関数では、return文は省略できる

呼び出し側では、関数から返された値を受け取ります。その最も簡単な方法は、呼び出し側に変数を用意して、関数からの戻り値を代入することです。呼び出し側は以下のように書きます。

【呼び出し側の書き方】

```
main()
{
        型      変数名；

        :
        :
        変数名　=　関数名();
        :
        :
}
型　関数名()
{
          :
          :
        return データ;
}
```

関数がvoid型のときは、関数名(); となります

2つの型は一致していなければならない

プログラムにはいろいろな書き方がありますが、C言語では、呼び出し側を先に書くのが一般的です。コンパイラは上から順にコンパイルしますから、呼

び出し側で関数の戻り値を代入する文をコンパイルするときには、関数の型がまだわからず、左辺の変数と右辺の関数の型が一致しているかどうか判断することができません。そこで、関数の型だけを先に書いておきます。これをプロトタイプ宣言といいます。

【プロトタイプ宣言の書き方】

```
#include <stdio.h>

型  関数名();      ← プロトタイプには;をつける

main()
{
        型    変数名;
         :
         :
        変数名  =  関数名();      ← 一致していなければならない
         :
         :
}
型  関数名()       ← 関数の定義には、;はつけない
{
      :
      :
      return データ;
}
```

●戻り値のある関数・ない関数

チーム名と選手数とを別々の関数で表示するプログラムを考えます。

【実行結果】

```
        FC希望ヶ丘  霧が丘FC    星が丘SS    夢ヶ丘SC
選手数    65          78          65          87
全部で   295
```

【プログラム例】

```
#include <stdio.h>

/*関数の宣言（プロトタイプ）*/
void    team_disp(void);          /*チーム名の表示*/
int     ninzu_disp(void);         /*選手数の表示*/

main()
{
        int     total;  /*合計人数*/

        /*チーム名の表示*/
        team_disp();

        /*選手数と総選手数の表示*/
        total = ninzu_disp();
        printf("全部で%6d¥n",total);
}

/**********************************************
        チーム名を表示
                戻り値：なし
**********************************************/
void    team_disp()
{
        char    team[4][14]={     "FC希望ヶ丘","霧が丘FC  ",
                                  "星が丘SS  ","夢ヶ丘SC  "};
        int     i;                /*添え字*/

        printf("      ");                     /*空白の表示*/
        for(i=0;i<4;i++)                      /*4チーム分*/
        {
                printf("%-12s",team[i]); /*チーム名を表示*/
        }
        printf("¥n");                         /*改行*/
}
```

- team_disp(); → 戻り値がないvoid型の関数呼び出し
- total = ninzu_disp(); → 関数の戻り値を受け取る変数totalを左辺に書く
- printf("全部で%6d¥n",total); → 関数ninzu_dispからの戻り値（合計人数）を表示
- } → 関数がvoid型で戻り値がないので、return文はなくてもよい

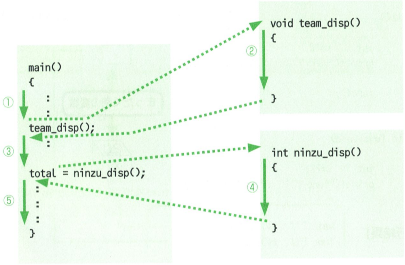
次ページに続く

```
/*******************************************************
        選手数を表示
                戻り値：人数の合計
*******************************************************/
int     ninzu_disp()
{
        int     ninzu[4]={65,78,65,87};  /*4チームの人数*/
        int     i;                       /*添え字*/
        int     gokei=0;                 /*合計人数*/

        printf("選手数");
        for(i=0;i<4;i++)                 /*4チーム分*/
        {
                printf("%6d      ",ninzu[i]);       /*人数を表示*/
                gokei = gokei + ninzu[i];           /*合計人数を求める*/
        }
        printf("¥n");

        return gokei;   /*合計人数を呼び出し側に戻す*/
}
```

このプログラムの実行の順序は、以下のようになります。



5-14 関数と変数の有効範囲（スコープ）

今まで使ってきた変数や配列には、実は、有効範囲があります。ブロックの中で宣言された変数や配列は、そのブロックの中でないと使うことができません。その反対に、すべての関数の外側で宣言された変数や配列は、すべての関数の中から参照したり、書き換えたりすることができます。

●ブロック有効範囲の変数

ブロックの内側で宣言された変数や配列の有効範囲を**ブロック有効範囲**といいます。別のブロックに同じ名前の変数や配列があると、まったく違うものとして扱われます。

【プログラム例】

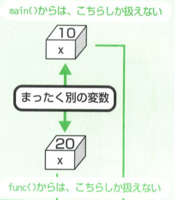
```
#include <stdio.h>

void func(void);        /*関数の宣言（プロトタイプ）*/

main()
{
        int     x=10;
        printf("mainでは、xの値は%d¥n",x);

        func();
}

void func(void)
{
        int     x=20;
        printf("funcでは、xの値は%d¥n",x);
}
```



【実行結果】

```
mainでは、xの値は10
funcでは、xの値は20
```

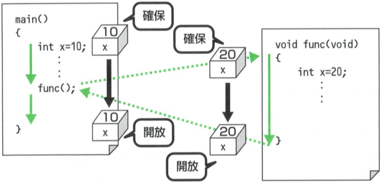
前節のプログラム（P.198）では、チーム名は、これを表示する関数team_disp内に、選手の人数は、これを表示する関数ninzu_disp内で宣言しています。ですからこれらの配列は、ブロック有効範囲となって、よその関数からは見ることができないのです。チーム名や選手数といった大切なデータが、このようにバラバラに宣言されるのは、本来は望ましいことではありません。

一方、添え字iは、両関数ともで宣言しています。2つの変数は名前は同じですが、異なる別のものとして扱われます。この変数iは、プログラムの都合上必要なものであって、チーム名や選手数のような大切なデータが保存されているわけではありません。このような変数は**ローカル変数**と呼ばれ、必要なところで、それぞれ宣言するのがよいとされています。

●ブロック有効範囲のライフサイクル

ブロック有効範囲の変数は、そのブロックを実行するときに確保され、ブロックの実行が終了すると、その領域は開放されます（注意：静的記憶域期間を指定するstaticがついているときを除く）。

左ページのプログラムの変数のライフサイクルは下図のようになります。



●ファイル有効範囲

すべての関数から使うことのできる変数や配列の有効範囲を、**ファイル有効範囲**といいます。ファイル有効範囲の変数や配列は、すべての関数の外側に書きます。

【プログラム例】

```
#include <stdio.h>

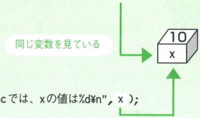
void func(void);        /*関数の宣言（プロトタイプ）*/

int     x=10;           /*グローバル変数の宣言*/

main()
{
        printf("mainでは、xの値は%d¥n", x );

        func();
}

void func(void)
{
        printf("funcでは、xの値は%d¥n", x );
}
```



【実行結果】

```
mainでは、xの値は10
funcでは、xの値は10
```

このような変数は**グローバル変数**とも呼ばれています。グローバル変数は、どの関数からも見ることができるという便利な反面、どの関数からも書き換えることができるため、プログラムのバグなどにより、誤ってデータを壊してしまう危険性も高くなります。本当に必要なものだけを厳選してグローバル変数にする配慮が必要です。前節のプログラムのチーム名や選手数のデータは、グローバル変数としてまとめて宣言してもよい性質のものです。

同じ名前のグローバル変数とローカル変数とを宣言することもできますが、扱いが紛らわしいので、あまりお勧めできません。

【誤ったプログラム例】

以下のプログラムは、0で初期化されたグローバルな配列xに、添え字と同じデータを代入するつもりのプログラムです。1つの要素に代入するたびに、代入の様子を表示しようと、関数funcを呼び出しています。ところが、main()とfunc()の両方でグローバル変数を使ってしまったので、1回目のfunc()の呼び出しで、iの値が5になってしまい、main()のループを終了してしまって、正しく実行することができません。変数iをローカル変数にすれば、うまく動きます。

```
#include <stdio.h>

void func(void);          /*関数の宣言（プロトタイプ）*/

int     x[5]={0,0,0,0,0};          /*グローバル変数の宣言*/
int     i;

main()
{
        for(i=0;i<5;i++)
        {
                x[i]=i;
                func();
        }
}

void func(void)
{
        for(i=0;i<5;i++)
        {
                printf("%5d",x[i]);
        }
}
```

iが0から4まで5回実行するはずが…

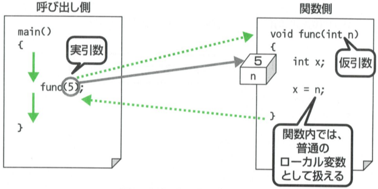
1回目の呼び出しで、
iの値が変わってしまう

5-15 関数に情報を渡す引数

呼び出し側が関数を呼び出すときに、「このデータについて処理をしてほしい」といった情報を関数に渡すことができれば、汎用的な関数を作ることができます。関数側は、呼び出し側からの情報を受け取るための変数を用意します。これを仮引数といいます。呼び出し側が渡す情報を実引数といいます。

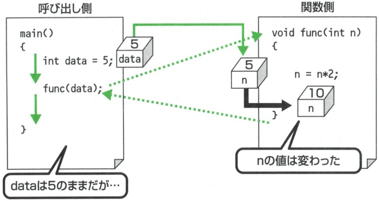
●実引数と仮引数

引数は、呼び出し側で持っているデータを関数側に渡すための手段です。たとえば、1からnまでの整数の和を求める関数では、nは具体的にいくつなのか、を知らなければ結果を出すことができません。これを呼び出し側から関数側に知らせることで、呼び出し側は、関数を制御することができるようになります。



呼び出し側で実引数として指定した値は、関数が呼び出されたときに、関数側の仮引数に代入されます。仮引数は、関数の中では、普通のローカル変数とまったく同様に扱われます。これは、関数のローカル変数ですから、呼び出し側では、仮引数の変数名（上の図ではn）を使うことはできません。

実引数として、呼び出し側の変数を指定することもできます。その場合は、関数が呼び出されたときに、変数に記録されているデータが仮引数に代入されます。



呼び出し側の実引数（上の図ではdata）と、関数側の仮引数（上の図ではn）とは、別のものですから、仮引数の値をいくら変更しても、呼び出し側には何の影響もありません。

●引数の書き方

関数側の仮引数は、変数ですから、型を明らかにしなければなりません。関数を定義する文の、関数名に続く( )内に、型と変数名と書きます。仮引数は、複数個指定することができます。その場合には「,」で区切って、並べて書きます。引数が1つもない関数は、( )内にvoidと書きます。

関数プロトタイプでも、引数の型を明らかにします。プロトタイプでは、仮引数の変数名は書かなくてもよいことになっていますが、関数の宣言文と同じ書き方をしておくことをお勧めします。

呼び出し側の実引数には、関数側の仮引数の型に合ったデータを、その順に書きます。

【引数の書き方】

```
#include <stdio.h>
 型  関数名(型  仮引数名,型  仮引数名…);
main()
{
        型            変数名;
        :
        :
        変数名  =     関数名(実引数,実引数,…);
        :
        :
}
  型  関数名(型  仮引数名,型  仮引数名…)
{
        :
        :
        return データ;
}
```

プロトタイプ宣言

型が一致していなければならない

関数呼び出し

関数定義

●データを渡す引数

1～nまでの整数の和を求める関数を作ります。main()で、いくつまでの和を求めるかをキーボードから入力し、これを実引数として、関数wa()に渡します。関数wa()は、1～nまでの整数の和を求め、これを戻り値として返します。これを受けて、main()では、下図のように表示します。

【実行結果】

```
整数を入力：10
1～10までの整数の和は55
```

の部分はキーボードからの入力

【プログラム例】

```
#include <stdio.h>

int wa(int n); /*1～nまでの整数の和を求める*/

main()
{
        int     n;        /*どこまでの和を求めるか*/
        int     y;        /*求めた和*/

        /*データの入力*/
        printf("整数を入力：");
        scanf("%d",&n);

        /*1～nまでの整数の和を求める*/
        y = wa(n);

        /*結果の表示*/
        printf("1～%dまでの整数の和は%d¥n",n,y);
}

/**************************************************
        1からnまでの整数の和を求める関数
                戻り値：求まった和
**************************************************/
int wa(int n)
        /* n : どこまでの和を求めるか*/
{
        int     i;                  /*ループ回数*/
        int     kekka=0; /*戻り値*/

        for(i=1;i<=n;i++)
        {
                kekka = kekka + i;
        }

        return kekka;
}
```

実引数と仮引数が同じであってもかまいません。この場合、呼び出し側のnと関数側のnは、異なる変数として扱われます

合計を求めるアルゴリズムの応用

関数内で宣言された変数kekkaの内容が戻り値として呼び出し側に渡されます。プログラムには書かれていませんが、
y = kekka; (yは呼び出し側の変数)
が行われたことになります

●引数が複数ある例

性別を表1のような記号で入力し、表示します。また、年齢を入力して表2に従って表示します。

表1

記号	表示
'M'	男子
'W'	女子
その他	不明

表2

学年	表示
1・2年	SLの部
3・4年	LLの部
5・6年	Lの部
その他	該当なし

【実行結果】

```
性別を入力：W
学年を入力：6
Lの部　男子
```

```
性別を入力：S
学年を入力：2
SLの部　不明
```

の部分はキーボードからの入力

【プログラム例】

```
#include <stdio.h>

/*関数のプロトタイプ宣言*/
void    disp_data(int n,char s);      ← int型、char型の順に2つの引数

main()
{
        int     gakunen; /*学年*/
        char    seibetu; /*性別*/

        printf("性別を入力：");
        scanf("%c",&seibetu);

        printf("学年を入力：");
        scanf("%d",&gakunen);         ← int型、char型の順に2つの引数

        disp_data(gakunen,seibetu);
}
```


次ページに続く

```
/*******************************************
          年齢に対する呼称と性別を表示
                    戻り値：なし
*******************************************/
void    disp_data(int n,char s)
                    /* n : 年齢*/
                    /* s : 性別*/
{
          /*年齢に対する呼称*/
          if(n == 1 || n == 2)
          {
                    printf("SLの部  ");
          }
          else if (n == 3 || n == 4)
          {
                    printf("LLの部  ");
          }
          else if (n == 5 || n == 6)
          {
                    printf("Lの部  ");
          }
          else
          {
                    printf("該当なし  ");
          }

          /*性別の表示*/
          if(s == 'M')
          {
                    printf("男子");
          }
          else if(s == 'W')
          {
                    printf("女子");
          }
          else
          {
                    printf("不明");
          }
}
```

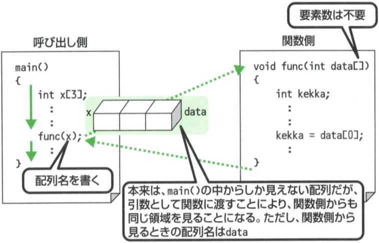
実引数のデータが、書かれた順に、
仮引数に入ります。
プログラムには書かれていませんが、
 n = gakunen;
 s = seibetu
が行われたのと同じです

5-16 配列を引数に渡す

配列には、要素の数だけデータが入っています。引数として、配列を指定すると、このたくさんのデータを一度に関数に渡すことができます。配列名は、配列の先頭のアドレスを表しますので、正確には、データを全部渡しているのではなく、先頭のアドレスを渡しています。

●配列の渡し方

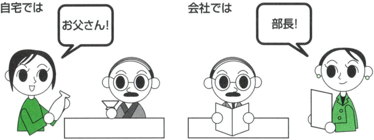
仮引数に配列を渡すには、受け手である関数側の仮引数に、配列を表す[ ]をつけます。1次元配列のときは、要素数は書きません。呼び出し側の実引数には、配列名を書きます。



引数として変数を渡すときと異なり、配列の要素のすべてが関数側に代入されるのではなく、先頭のアドレスだけが関数側に渡されます。つまり、関数側では、data[ ]という名前で、同じ場所を見ているのです。人間にたとえると、お父さんは、会社に行けば「部長」と呼ばれるけれど、家に帰ると「お父さん」と呼ばれる、でも、「部長」も「お父さん」も同一人物である、というの

に似ています。このメモリ領域は、呼び出し側からは「x」と呼ばれ、関数側からは「data」と呼ばれるのです。

▼呼ばれる側によって、呼称が変わる



配列が2次元のときは、関数側の仮引数には、1行の要素数だけは書く必要があります。それがないと、どこまでが1行なのかわからないからです。

●配列を引数として渡すプログラム

5-13で紹介したプログラム（P.198）では、必要なデータをそれぞれの関数の中で定義しました。引数として配列を渡すことができると、大切なデータを`main()`の中にまとめて定義することができます。

また、配列名のほかに、配列要素数を別の引数として関数側に渡すようにすると、要素数がいくつの配列であっても対応できる関数となり、汎用性が高くなります。

【実行結果】

```
        FC希望ヶ丘  霧が丘FC    星が丘SS    夢ヶ丘SC
選手数     65          78          65          87
全部で    295
```

【プログラム例】

配列名の他にデータの個数も引数として与える

```
#include <stdio.h>

/*関数の宣言*/
void    team_disp(char name[][14],int n);
int     ninzu_disp(int x[],int n);

main()
{
        /*データの定義*/
        char    team[4][14]={   "FC希望ヶ丘","霧が丘FC   ",
                                "星が丘SS   ","夢ヶ丘SC   "};
        int     ninzu[4]={65,78,65,87};         /*人数*/

        int     total;

        /*チーム名の表示*/
        team_disp(team,4);
```

配列名はteam、データの個数は4個

```
        /*選手数と総選手数の表示*/
        total = ninzu_disp(ninzu,4);
        printf("全部で%6d¥n",total);
}
```

配列名はninzu、データの個数は4個

```
/**********************************
        チーム名を表示
**********************************/
void    team_disp(char name[][14],int n)
{
        int     i;

        printf("      ");
        for(i=0;i<n;i++)
        {
                printf("%-12s",name[i]);
        }
        printf("¥n");
}
```

n個の文字列を表示する関数

次ページに続く

```
/*********************************************
        選手数を表示
            戻り値：データの合計
*********************************************/
int     ninzu_disp(int x[],int n)
{
        int     i;
        int     gokei=0;

        printf("選手数");
        for(i=0;i<n;i++)
        {
                printf("%6d      ",x[i]);
                gokei = gokei + x[i];
        }
        printf("¥n");

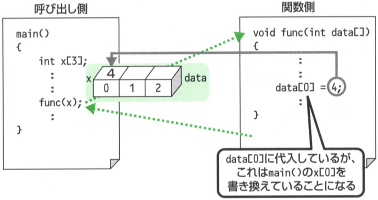
        return gokei;
}
```

n個の整数を表示する

n個の整数を合計する

●引数としての配列を書き換える

仮引数として渡された配列は、呼び出し側と同じ領域に異なる名前をつけたものですから、関数側から、その内容を直接書き換えることができます。



5-9で学習したchar型配列に入っている文字列を、別のchar型配列に複写するプログラムを関数を使って書き直します。引数として、元の文字列が入っている配列と複写するための別の配列を渡します。

【プログラム例】

```
#include <stdio.h>

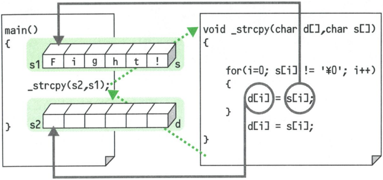
/*関数プロトタイプ*/
void _strcpy(char d[],char s[]);

main()
{
        /*配列の宣言*/
        char    s1[10]="Fight!";
        char    s2[10];

        _strcpy(s2,s1);              /*関数の呼び出し*/

        /*結果の表示*/
        printf("s1の文字列は%s¥n",s1);
        printf("s2の文字列は%s¥n",s2);
}
/*********************************************
        文字列を複写する関数
*********************************************/
void _strcpy(char d[],char s[])
{
        int     i;      /*添え字*/

        for(i=0;s[i] != '¥0';i++)
        {
                d[i] = s[i];
        }
        d[i] = s[i];
}
```



呼び出し側では、s1, s2という名前で呼ばれている配列が、s1, s2を引数に渡すことで、関数側では、s, dという名前で呼ばれます。

```
d[i] = s[i];
```

上の文は、s[i]の内容を取り出してd[i]に書き込むものですが、これは、呼び出し側のs1[i]の内容を取り出してs2[i]に書き込んでいることになります。ですからこの関数は、戻り値はありませんが、直接、値を書き込むことで、「値を戻している」といえます。関数の戻り値では、1つのデータしか戻すことができません。関数側から呼び出し側にたくさんのデータを渡したいときには、配列を介して渡すこの方法は、有効な方法です。

●グローバル変数を用いたデータの受け渡し

一方、配列をすべての関数の外側で宣言し、グローバル変数にした場合には、グローバル変数は、呼び出し側からも関数側からも直接書き込むことができますので、引数を受け渡す必要がなくなります。一見、便利なようですが、常に決められた配列にデータを入れなければならない、あるいは、常に決められた配列にデータが書き込まれることになりますので、関数の汎用性・独立性は、ともに低下します。

5-17 ポインタを引数に渡す

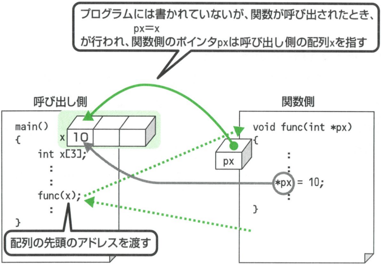
配列名は、先頭の要素のアドレスを持ちます。「配列名を渡す」ということは、「その配列を指すポインタを渡す」ことに他なりません。前節では、関数内の処理を配列で書きましたが、仮引数をポインタで受け取り、ポインタを使って同様の処理を書くことができます。

●ポインタの渡し方

関数にアドレスを渡すとき、仮引数をポインタ型で宣言します。この場合、実引数には、配列名を書くこともできますし、変数や配列を指すポインタを書くこともできます。

【実引数に配列名を指定した場合】

関数が呼び出されたとき、実引数で指定された配列の先頭のアドレスが仮引数pxに代入されます。



関数内で宣言された変数や配列は、その関数の中でしか参照することはできません。しかし、関数の引数にポインタを与えることにより、メモリ内の番地（アドレス）という、共通して知ることのできる情報に変換でき、結果として、他の関数内で宣言された変数や配列も参照できることになります。

●ポインタを引数として渡すプログラム

5-16では、文字列を複写する関数_strcpyを、配列で書きました。今度は、関数の処理をポインタで書き換えてみましょう。

【プログラム例】

```
#include <stdio.h>

void    _strcpy(char *pd,char *ps);

main()
{
        char    s1[10] = "Fight!";        /*元の文字列*/
        char    s2[10];                   /*複写先*/

        _strcpy(s2,s1);                   /*複写する*/

        printf("s1の文字列は%s¥n",s1);
        printf("s2の文字列は%s¥n",s2);
}

/****************************************************
        文字列を複写する関数
****************************************************/
void    _strcpy(char *pd,char *ps)
{
        for(;*ps != '¥0';ps++,pd++)
        {
                *pd = *ps;        /*複写*/
        }
        *pd = *ps;
}
```

（_strcpy(s2,s1); への注記）配列名はアドレスを持つので、実引数に配列名が書かれると…

（void _strcpy(char *pd,char *ps) への注記）仮引数には、そのアドレスが入る

5-18 自分を呼び出す再帰関数

関数の中からは、別の関数を呼び出すことができますが、これが、別の関数ではなく、自分自身であるものを再帰関数といいます。求めた値を使って、次の値を求める場合や、次々に分割していくような場合に、再帰を使うと簡単に実現することができます。

●再帰関数を使うとよい例

整数の階乗を求めます。階乗とは、1からその整数までを順に掛けた積で、整数nの階乗はn!と書きます。

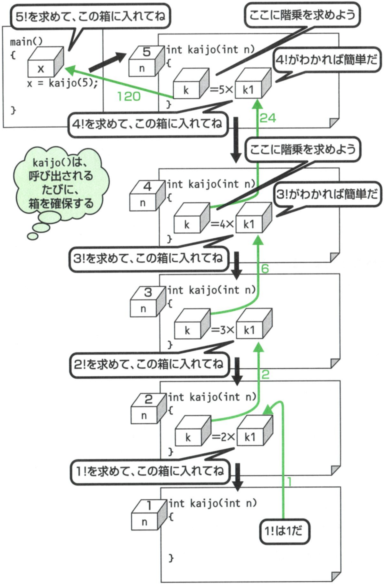
例　5!＝5×4×3×2×1＝120

5の階乗を求めるとき、先頭から順に掛け算をしても求まりますが、次のように考えると、もっとスマートに解決することができます。

① 5!は、4!に5を掛けたものである　　5!＝4!×5
　したがって、4!が求まれば5!は求まる。
② 4!は、3!に4を掛けたものである　　4!＝3!×4
　したがって、3!が求まれば4!は求まる。
③ 3!は、2!に3を掛けたものである　　3!＝2!×3
　したがって、2!が求まれば3!は求まる。
④ 2!は、1!に2を掛けたものである　　2!＝1!×2
　したがって、1!が求まれば2!は求まる。
⑤ 1!は、1である。

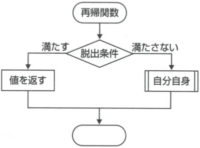
このように、階乗を求めたい数より1少ない数の階乗を求めることを、自分自身に要求し続けるのです。そして、1!を求めることになったとき、答えは1であり、これ以上要求する必要がなくなります。これを再帰の脱出条件といいます。

▼階乗の求め方


5!を求めて、この箱に入れてね
main()
{
x
x = kaijo(5);
}
120
5
n
int kaijo(int n)
{
k
=5×
k1
}
ここに階乗を求めよう
4!がわかれば簡単だ
4!を求めて、この箱に入れてね
24
kaijo()は、呼び出されるたびに、箱を確保する
ここに階乗を求めよう
4
n
int kaijo(int n)
{
k
=4×
k1
}
3!がわかれば簡単だ
3!を求めて、この箱に入れてね
6
3
n
int kaijo(int n)
{
k
=3×
k1
}
2!を求めて、この箱に入れてね
2
2
n
int kaijo(int n)
{
k
=2×
k1
}
1!を求めて、この箱に入れてね
1
1
n
int kaijo(int n)
{
1!は1だ
}

●再帰関数の形式

再帰関数では、これ以上自分自身を呼ばない、という脱出条件が必ず必要になります。そうでないと、永久に自分自身を呼び出して、無限ループに陥ってしまいます。ですから、再帰関数は、一般に以下のような流れ図になります。



●階乗を求めるプログラム

階乗を求めるプログラムを、再帰関数を使って書くと、以下のようになります。

【プログラム例】

```
#include <stdio.h>

/*関数のプロトタイプ宣言*/
int     kaijo(int n);

main()
{
        int     x;

        x = kaijo(5);           /*5の階乗を求める*/

        printf("5の階乗は%dです¥n",x);
}
```

次ページに続く

```
/*********************************************
        階乗を求める再帰関数
*********************************************/
int     kaijo(int n)
{
        int     k;          /*求める階乗*/
        int     k1;         /*1つ前の階乗*/

        if (n == 1)         /*脱出条件*/
        {
                /*満たすとき*/
                k = 1;              /*階乗は1*/
        }
        else
        {
                /*満たさないとき*/
                k1 = kaijo(n-1);            /*1つ前の階乗を要求*/
                k = n *k1;                  /* n×(n-1)! で階乗は求まる*/
        }

        return k;
}
```

【実行結果】

```
5の階乗は120です
```

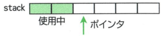
One Point 用語解説

➡スタック

再帰関数では、自分自身が呼び出されるたびに、箱（変数）が用意されます。これは、いったいどういうしくみになっているのでしょうか？

関数の呼び出しには、スタックと呼ばれるメモリ領域が深く関わっています。関数呼び出しでは、関数の実行終了後、呼び出した命令に制御が戻ります。つまり、関数を呼び出す際には、呼び出した命令が記憶されている番地を覚えておかなければなりません。特に再帰関数の場合は、何回呼ばれるかわかりませんので、呼んだ回数分を記録する必要があります。

そこで、何でも書き込める、多めの領域を確保しておきます。そして、関数内で確保された変数や、関数を呼び出した命令のアドレスなど、関数を実行している間に記録しておかなければならない情報を発生した順に記録します。このとき、どこまで使っているかを示すポインタを用意しておきます。書き込むときは、ポインタの指す先へ、読み出すときは、ポインタの指す先から、読み出します。つまり、最後に入れた情報を最初に取り出すことになります。このようなしくみのメモリ領域をスタックといいます。



5-19 ライブラリ関数を便利に使う

よく使われる関数を、そのつど、それぞれに作っていたのでは、効率が悪いでしょう。誤りのない保証された関数を作っておき、使いたいときに、誰でもが自由に使えると、開発効率がアップするばかりでなく、信頼性・汎用性も向上します。このような関数をライブラリ関数といいます。

●提供されるライブラリ関数

「ライブラリ」とは、英語で図書館という意味です。図書館には本がたくさんありますが、コンパイラと一緒に提供されるC言語のライブラリには、便利な関数がたくさん詰まっていて、必要なときに使えるようになっています。多くのライブラリ関数が、JIS規格により規定されており、この規定にしたがって呼び出しさえすればよいのです。

●ライブラリ関数の使い方

ライブラリ関数の呼び出し方は通常の関数とまったく同じですが、関数本体は、ライブラリの中に用意されていますので、ソースプログラムに改めて書く必要はありません。通常の関数では、呼び出しに先立って関数や引数の型を明らかにするプロトタイプ宣言を書きますが、ライブラリ関数では、プロトタイプ宣言はヘッダと呼ばれるファイルに収められており、`#include`を使ってヘッダファイルをソースプログラム中に取り込めばよいようになっています。今まで、おまじないのように書いてきた、

```
#include <stdio.h>
```

は、実は、ライブラリ関数を呼び出すためのヘッダファイルの取り込みだったのです。そして、表示をするために使ってきた`printf()`などは、実は、情報を画面に表示する機能を持ったライブラリ関数だったのです。

C言語のソースファイルの拡張子は「`.c`」ですが、ヘッダファイルの拡張子

は「.h」です。JIS規格ではライブラリ関数の機能別に15個のヘッダファイルがコンパイラから提供されることになっていて、どのライブラリ関数を使うときに、どのヘッダファイルをインクルードしなければならないかが決まっています。

【ふつうの関数】

```
/*プロトタイプ宣言*/
型　関数名

main()
{
    :
    :
   関数名();
   :
   :
}
  型　関数名()
{
        関数本体の処理
}
```

関数呼び出し

【ライブラリ関数】

```
#include <ヘッダファイル名>
main()
{
    :
    :
   関数名();
    :
    :
}
```

プロトタイプ宣言をするかわりにヘッダファイルを取り込む

関数呼び出し

関数本体の処理は書かなくてよい

1つ1つのライブラリ関数の仕様は、コンパイラのマニュアルに詳しく書かれています。

JISで規定されているヘッダファイルは以下のとおりです。

assert.h	locale.h	stddef.h
ctype.h	math.h	stdio.h
errno.h	setjmp.h	stdlib.h
float.h	signal.h	string.h
limits.h	stdarg.h	time.h

5-20 文字列を処理するライブラリ関数を使う

文字列操作は、実際の業務でも欠かすことのできない項目ですから、文字列を扱うライブラリ関数は、多種用意されています。これらを上手に使うことで、煩雑な文字列操作を効率よく行うことができます。ここでは、よく使われる文字列関数を紹介し、それらを使った応用例に挑戦します。

●文字列の長さを求める関数　～ strlen

[ヘッダ]　string.h

[形　式]　size_t　strlen(const char *s);

[引　数]　s：文字列を指すポインタ または 文字列が格納されている配列名

[返却値]　'¥0'を除く文字列の長さを返す

[プログラム例]

```
#include <stdio.h>
#include <string.h>

main()
{
        char    a[20]="Happy!";             /*文字列*/
        int     na;                         /*文字数*/

        na = strlen(a);
        printf("文字数は%d¥n",na);
}
```

[実行結果]

```
文字数は6
```

(注意1) size_tは、変数のサイズを表すために定義された型で、符号なし整数型と同じです。

(注意2) constは、関数を実行してもその配列の内容に変化がないことを表しています。

●文字列を複写する関数　～ strcpy

[ヘッダ]　string.h

[形　式]　char strcpy(char *s1,const char *s2);

[引　数]　s1：複写先の文字型配列へのポインタ　または　配列名
　　　　　s2：複写元の文字型配列へのポインタ　または　配列名

[返却値]　s1を返す（複写先の文字型配列の先頭のアドレス）

[プログラム例]

```
#include <stdio.h>
#include <string.h>

main()
{
        char    a[20]="Happy!";          /*複写元の文字列*/
        char    b[20];                   /*複写先*/
        char    *pb;                     /*ポインタ*/

        pb = strcpy(b,a);
        printf("配列aの内容は    %s¥n",a);
        printf("配列bの内容は    %s¥n",b);
        printf("関数の戻り値は    %p¥n",pb);
        printf("配列bのアドレスは%p¥n",b);
}
```

[実行結果]

```
配列aの内容は      Happy!
配列bの内容は      Happy!
関数の戻り値は     0012FF64
配列bのアドレスは  0012FF64
```

文字列が複写されて
2つの配列の内容は
同じになった

関数の戻り値は、複写先の配列の先頭のアドレスに等しい
（注意）この値は、環境によって、異なります。関数の戻り値と配列bの
アドレスとが等しいことを確認してください

●文字列を指定文字数だけ複写する関数　～ strncpy

[ヘッダ]　string.h

[形　式]　char strncpy(char *s1,const char *s2,size_t n);

[引　数]　s1：複写先の文字型配列へのポインタ　または　配列名

s2：複写元の文字型配列へのポインタ　または　配列名

n：複写を行う最大文字数

[返却値]　s1を返す（複写先の文字型配列の先頭のアドレス）

[プログラム例]

```
#include <stdio.h>
#include <string.h>

main()
{
        char    a[20]="Happy!";             /*複写元の文字列*/
        char    b[20]="1234567890";         /*複写先*/
        char    c[20]="1234567890";         /*複写先*/

        strncpy(b,a,3);          /*先頭から3文字だけ複写*/
        strncpy(&c[5],&a[1],4); /*1番目から4文字だけ複写*/
        printf("配列aの内容は    %s¥n",a);
        printf("配列bの内容は    %s¥n",b);
        printf("配列cの内容は    %s¥n",c);
}
```

[実行結果]

```
配列aの内容は    Happy!
配列bの内容は    Hap4567890
配列cの内容は    12345appy0
```

先頭から3文字分だけ複写されている

1文字目から4文字分だけ複写されている

●文字列を比較する関数 ～ strcmp

[ヘッダ] string.h

[形　式] char strcmp (const char *s1,const char *s2);

[引　数] s1：複写先の文字型配列へのポインタ　または　配列名
s2：複写元の文字型配列へのポインタ　または　配列名

[返却値] 2つの文字列を比較し、辞書順でどちらが前になるかを、以下のように返す
s1が指す文字列　<　s2が指す文字列　のとき　負の値
s1が指す文字列　=　s2が指す文字列　のとき　0
s1が指す文字列　>　s2が指す文字列　のとき　正の値

[プログラム例]

```
#include <stdio.h>
#include <string.h>

main()
{
        char    a[20];              /*文字列*/
        char    b[20];              /*文字列*/
        int     kekka;              /*比較の結果*/

        /*キーボードから入力*/
        printf("1つ目の文字列は？  ：");
        scanf("%s",a);
        printf("2つ目の文字列は？  ：");
        scanf("%s",b);

        kekka = strcmp(a,b);    /*比較！！*/

        /*結果の表示*/
        if      (kekka < 0) printf("a < b¥n");
        else if (kekka == 0)printf("a = b¥n");
        else                printf("a > b¥n");
}
```

[実行結果1]

```
1つ目の文字列は？　：aa
2つ目の文字列は？　：aaa
a < b
```

[実行結果2]

```
1つ目の文字列は？　：aaa
2つ目の文字列は？　：aaa
a = b
```

[実行結果3]

```
1つ目の文字列は？　：aba
2つ目の文字列は？　：aaa
a > b
```

キーボードから入力された文字列

●文字列を指定文字数だけ比較する関数　～ strncmp

[ヘッダ]　string.h

[形　式]　char strncmp(const char *s1,const char *s2,size_t n);

[引　数]　s1：複写先の文字型配列へのポインタ　または　配列名

s2：複写元の文字型配列へのポインタ　または　配列名

n：比較する文字数

[返却値]　2つの文字列を先頭からn文字だけ比較し、辞書順でどちらが前になるかを、以下のように返す。文字列の長さがn文字未満のときは、文字列の最後までを比較する。

s1が指す文字列　＜　s2が指す文字列　のとき　負の値

s1が指す文字列　＝　s2が指す文字列　のとき　0

s1が指す文字列　＞　s2が指す文字列　のとき　正の値

[プログラム例]

```
#include <stdio.h>
#include <string.h>

main()
{
        char    a[20];            /*文字列*/
        char    b[20];            /*文字列*/
        int     kekka;            /*比較の結果*/

        printf("1つ目の文字列は？  ：");
        scanf("%s",a);
        printf("2つ目の文字列は？  ：");
        scanf("%s",b);

        kekka = strncmp(a,b,3); /*比較！！*/

        if      (kekka < 0) printf("a < b¥n");
        else if (kekka == 0)printf("a = b¥n");
        else                printf("a > b¥n");
}
```

[実行結果]

```
1つ目の文字列は？  ：aaaa
2つ目の文字列は？  ：aaab
a = b
```

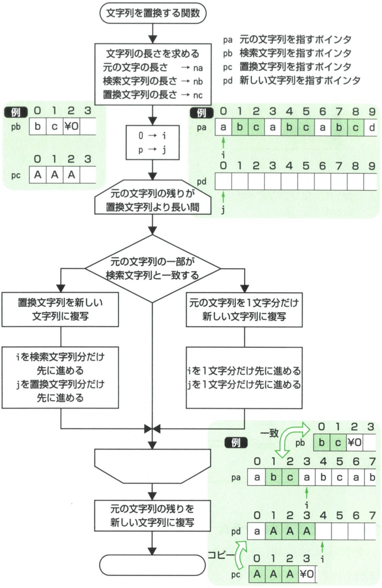
先頭の3文字だけが比較の対象となる

□の部分はキーボードからの入力

●文字列を置換するプログラム

長い文字列の中から短い文字列を検索し、別の文字列に置換して、新しい長い文字列を作る関数を作ります。ライブラリ関数を用いると、詳細をライブラリ関数に任せることができるので、処理全体のアルゴリズムだけを考えればよいようになります。

[流れ図]


文字列を置換する関数
文字列の長さを求める
元の文字の長さ → na
検索文字列の長さ → nb
置換文字列の長さ → nc
pa 元の文字列を指すポインタ
pb 検索文字列を指すポインタ
pc 置換文字列を指すポインタ
pd 新しい文字列を指すポインタ
例
pb b c ¥0
pc A A A
0 → i
p → j
pa a b c a b c a b c d
i
pd
j
元の文字列の残りが置換文字列より長い間
元の文字列の一部が検索文字列と一致する
置換文字列を新しい文字列に複写
元の文字列を1文字分だけ新しい文字列に複写
iを検索文字列分だけ先に進める
jを置換文字列分だけ先に進める
iを1文字分だけ先に進める
jを1文字分だけ先に進める
一致
pb b c ¥0
pa a b c a b c a b
pd a A A A
コピー
pc A A A ¥0
元の文字列の残りを新しい文字列に複写

【プログラム例】

```
/***************************************
        文字列を置換する関数
***************************************/
void    stringChange(char *pa,char *pb,char *pc,char *pd)
                /* pa    元の文字列*/
                /* pb    検索文字列*/
                /* pc    置換文字列*/
                /* pd    置換した結果でき上がった文字列*/
{
        int     na,nb,nc;                       /*文字列の長さ*/
        int     i,j;                            /*添え字*/

        /*文字列の長さを求める*/
        na = strlen(pa);
        nb = strlen(pb);
        nc = strlen(pc);

        /*paの先頭より、pbをpcに置換しながらpdにコピーする*/
        for(i=0,j=0;i<na-nb;i++,j++)
                /* aの残りがbより短くなると、もう一致しない*/
        {
                if(strncmp(pa+i,pb,nb) == 0)
                        /*部分文字列がbと一致しているかどうか*/
                {
                        /*見つかった*/
                        strcpy(pd+j,pc);
                        i += nb - 1;
                        j += nc - 1;
                }
                else
                {
                        /*見つからない*/
                        *(pd+j) = *(pa+i);
                }
        }

        strcpy(pd+j,pa+i);          /*最後に残った文字列を複写*/
}
```

5-21 文字列数値変換ライブラリ関数を使う

キーボードやファイルからデータを入力した場合など、数値情報が「数字（文字）」のら列で表現されることがあります。これらのデータを演算する場合には、「数値」でなければなりません。「数字」を「数値」に変換するライブラリ関数を使って、演算できるようにします。

●文字列をint型の数値に変換する関数　～ atoi

［ヘッダ］　stdlib.h

［形　式］　int atoi(const char *s1);

［引　数］　s1：変換したい文字列へのポインタ　または　配列名

［返却値］　変換した結果のint型数値

文字列の中にint型の数値として適さない文字がある場合には、その文字の前までを変換した結果を返す

［プログラム例］

```
#include <stdio.h>
#include <stdlib.h>

main()
{
        char    a[20]="-12a3";  /*文字列*/
        int     na;             /*変換した数値*/

        na = atoi(a);
        printf("変換された数値は%d¥n",na);
}
```

［実行結果］

```
変換された数値は-12
```

'a'から後ろは無視される

●文字列をdouble型の数値に変換する関数　～ atof

[ヘッダ]　stdlib.h

[形　式]　int atof(const char *s1);

[引　数]　s1：変換したい文字列へのポインタ　または　配列名

[返却値]　変換した結果のdouble型数値
文字列の中にdouble型の数値として適さない文字がある場合には、その文字の前までを変換した結果を返す

[プログラム例]

```
#include <stdio.h>
#include <stdlib.h>

main()
{
        char    a[20]="-.12a3";          /*文字列*/
        char    b[20]="123.4e-1";        /*文字列*/
        double  da,db;                   /*変換した数値*/

        da = atof(a);
        db = atof(b);
        printf("変換された数値は%f¥n",da);
        printf("変換された数値は%f¥n",db);
}
```

[実行結果]

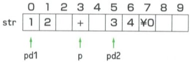
```
変換された数値は-0.120000
変換された数値は12.340000
```

整数部が0であり、1の位の数値が省略されていても対応できる

指数の形式にも対応できる
$123.4 \times 10^{-1} = 12.34$

●文字列を変換して演算するプログラム

文字型配列に格納された数式を解析して演算結果を求めるプログラムを考えます。ただし、数式の構文を解析する部分は、ここでは省略し、演算子・データを指すポインタを与えることとします。



また、有効な演算子は、加算（+）・減算（-）・乗算（*）・除算（/）とし、これ以外が指定された場合には、

演算子が違います

と表示します。除算については、除数が0の場合には、

0で割り算できません

と表示します。

【プログラム例】

```
/******************************************************************
      演算を行うプログラム
******************************************************************/
#include <stdio.h>
#include <stdlib.h>

main()
{
        char   str[20]="12 + 45";           /*文字列*/
        char   *pd1,*pd2,*p;                /*ポインタ*/
        int    x1,x2;                       /*データ*/

        pd1 = str;                          /*data1へのポインタ*/
        p   = &str[3];                      /*演算子へのポインタ*/
        pd2 = &str[5];                      /*data2へのポインタ*/
```

次ページに続く

```
        x1 = atoi(pd1);                         /*データ1*/
        x2 = atoi(pd2);                         /*データ2*/

        /*演算*/
        switch(*p)
        {
                case    '+':
                        printf("%s = %d¥n",str, x1 + x2);
                        break;
                case    '*':
                        printf("%s = %d¥n",str, x1 * x2);
                        break;
                case    '-':
                        printf("%s = %d¥n",str, x1 - x2);
                        break;
                case    '/':
                        if (x2 == 0)
                        {
                                printf("0で割り算できません¥n");
                        }
                        else
                        {
                                printf("%s = %d¥n",str, x1 / x2);
                        }
                        break;
                default:
                        printf("演算子が違います¥n");
        }
}
```

【実行結果】

```
12 + 45 = 57
```

5-22 メモリ操作ライブラリ関数を使う

大きな情報を記憶したい場合や、プログラムを実行してみなければどれくらいの記憶域が必要なのかを決定できない場合には、必要に応じてメモリを確保し、不要になったらこれを開放することができます。このようなメモリ操作のためのライブラリ関数も用意されています。

●メモリを確保する関数　～ malloc

[ヘッダ]　stdlib.h

[形　式]　void *malloc(size_t size);

[引　数]　size：確保したいバイト数

[返却値]　確保したメモリへのポインタ。確保できないときはNULL

[プログラム例]

```
#include <stdio.h>
#include <stdlib.h>

main()
{
        int     *p1;                    /*int型を指すポインタ*/

        if((p1 = malloc(sizeof(int))) != NULL)  /*メモリの確保*/
        {
                *p1 = 10;                       /*データの記録*/
                printf("p1の内容は　%d¥n",*p1);  /*表示*/
        }
        else
        {
                printf("メモリが確保できませんでした¥n");
        }

}
```

●メモリを開放する関数　～ free

[ヘッダ]　stdlib.h

[形　式]　void free(void *ptr);

[引　数]　ptr：開放したい領域へのポインタ。この領域は、mallocなどのメモリ管理ライブラリ関数で確保されたものであることが前提

[返却値]　なし

[プログラム例]

```
#include <stdio.h>
#include <stdlib.h>

main()
{
        char    *p;                     /*char型を指すポインタ*/
        int     i;                      /*添え字*/

        if((p = malloc(sizeof(char)*100)) != NULL)     /*メモリの確保*/
        {
                strcpy(p,"happy!");             /*データの記録*/
                printf("pの内容は　%s¥n",p);     /*表示*/
                free(p);                        /*メモリの開放*/
        }
        else
        {
                printf("メモリが確保できませんでした¥n");
        }
}
```

[実行結果]

```
pの内容は　happy!
```

●件数が未定のプログラム例

データの入力に先立って、データの件数を入力し、必要なだけのメモリを確保します。ここでは、処理は行わず、入力されたデータを表示するだけとします。

【実行結果】

```
処理件数を入力してください：8
0件目のデータを入力してください：10
1件目のデータを入力してください：20
2件目のデータを入力してください：30
3件目のデータを入力してください：40
4件目のデータを入力してください：50
5件目のデータを入力してください：60
6件目のデータを入力してください：70
7件目のデータを入力してください：80
入力データは以下のとおりです
    10    20    30    40    50    60    70    80
```

【プログラム例】

```
/**********************************************************
        メモリを確保して処理を行う
**********************************************************/
#include <stdio.h>
#include <stdlib.h>

main()
{
        int     *p;                     /*int型を指すポインタ*/
        int     n;                      /*処理件数*/
        int     i;                      /*添え字*/

        /*件数入力*/
        printf("処理件数を入力してください：");
        scanf("%d",&n);

        /*メモリの確保*/
```

次ページに続く

```
    if((p = malloc(sizeof(int)*n)) != NULL)    /*メモリの確保*/
    {
        /*データ入力*/
        for(i=0;i<n;i++)
        {
            printf("%d件目のデータを入力してください：",i);
            scanf("%d",(p+i));
        }

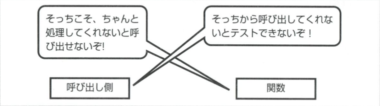
        /*データの表示*/
        printf("入力データは以下のとおりです¥n");
        for(i=0;i<n;i++)
        {
            printf("%5d",*(p+i));
        }

        /*メモリの開放*/
        free(p);
    }
    else
    {
        printf("メモリが確保できませんでした¥n");
    }
}
```

COLUMN

関数のテスト

大規模システムでは、プログラムを全部一人で作ることはできません。プロジェクトチームを作って分担し、同時に複数のプログラマが開発にあたることになります。そのような場合には、関数を作る人と、その関数を呼び出す部分を作る人とが異なるということもあるのです。関数は、適当な実引数を与えられて、呼び出されなければ実行できません。つまり、呼び出し側ができていないとテストができないことになります。ところが、呼び出し側も、関数で正しく処理をしてくれなければ次に進むことはできません。つまり、呼び出し側にとっても、関数ができていないとテストできないことになってしまいます。



そこで、適当な実引数を与えて関数を呼び出すだけのダミーのモジュールを作り、これを使って関数のテストを行います。これをドライバといいます。

また、処理をしたことにして結果だけを与えるダミーの関数を作り、これを呼び出すことで呼び出し側のテストを行います。これをスタブ（埋め草）といいます。



コンピュータを使う最大の目的は、情報を加工したり保存したり、必要に応じて取り出したり、といった情報処理だといえるでしょう。多くの場合、似たようなデータがたくさん集まって「情報」を形成します。

6章 構造体とファイル処理

最後のこの章では、1件分のデータをひとまとめに扱う構造体と、データを保存するためのファイル処理を学びます。

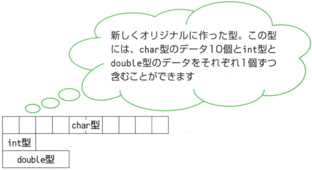
6-1 構造体でデータをまとめて扱う

配列は、多くのデータを一度に扱うことができましたが、すべて同じ型でなければなりません。ですから、異なる型のデータを1件分としてひとまとめにすることができません。そこで、異なる型の変数や配列を組み合わせ、これに名前をつけて、まとめて扱えるようにしたものを構造体といいます。

●構造体とは

たとえば個人情報を扱う場合、氏名はchar型、年齢はint型、身長はdouble型がよいでしょう。これを一人分としてワンパックにし、全体を新しい「型」として定義したものを**構造体**といいます。構造体の宣言は、新しい「型」の宣言であり、メモリ領域は確保されません。新しい型を使って変数や並列の宣言をしたとき、初めてメモリ領域が確保されます。つまり、構造体の宣言は、洋服を作るときの「型紙」を決めるようなものです。このように、基本の型を組み合わせた新しい型を**派生型**といいます。

▼派生型



●構造体の宣言

構造体を使うときは、まず、その構造体がどんな型の変数や配列を含むのか

という、構造体の形を宣言します。構造体を構成する各変数や配列を**構造体メンバ**といいます。構造体内で、お互いを区別するために、各メンバにつけた名前を**メンバ名**といいます。

【構造体の宣言】

```
struct  構造体タグ{
    型     メンバ名;
    型     メンバ名;
              :
              :
};
```

【プログラム例】

```
struct kojin_data{
        char    name[10];
        int     age;
        double  length;
}
```



●構造体型変数の定義

int型のデータを記録するためには、int型の大きさのメモリ領域を確保するために変数を宣言しました。構造体の場合も同様に、構造体型1個分のデータを保存するためには、その構造体型の変数を宣言して、メモリを確保します。

【構造体型変数の宣言】

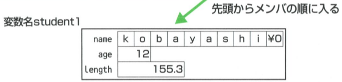
```
struct  構造体タグ    変数名={初期化子の並び};
```

【プログラム例】

```
/************************************************
        構造体変数の宣言
************************************************/
#include <stdio.h>

/*構造体の宣言*/
struct kojin_data{
        char    name[10];       /*氏名*/
        int     age;            /*年齢*/
        double  length;         /*身長*/
};

main()
{
        struct kojin_data       student1 = {"kobayashi",12,155.3};
}
```



typedef

構造体型の変数や配列を宣言するときは、それが構造体型であることがわかるように、構造体タグの前にstructをつけます。これでは型の名前が長いので、structまで含めた型の名前を決めることができます。構造体タグ名や変数名は、通常小文字を使いますが、typedefを含めた型の名前は、慣例的にすべて大文字にします。

【型名の宣言】

```
typedef struct  構造体タグ{
    型    メンバ名;
    型    メンバ名;
          :
          :
}型名;
```

【プログラム例】

```
/**********************************************
          構造体変数の宣言
**********************************************/
#include <stdio.h>

/*構造体の宣言*/
typedef struct kojin_data{
        char    name[10];         /*氏名*/
        int     age;              /*年齢*/
        double  length;           /*身長*/
}KOJIN;

main()
{
        KOJIN   student1 = {"kobayashi",12,155.3};
}
```

6-2 構造体型変数と演算子

構造体は、複数のデータをまとめて扱うものですから、まとめて別の変数に代入することができます。逆に、構造体型変数内の1つ1つのデータを別々に扱い、演算したり、データを画面に表示したりすることもできます。メンバ1つ1つに対する演算は、通常の演算と同様です。

●メンバを別々に扱う構造体メンバ演算子

構造体型変数内のある1つのデータを指定するには、メンバ演算子「.」を使い、以下の形式で書きます。

```
構造体型変数名.メンバ名
```

メンバ演算子を使うと、構造体型変数内の特定のメンバに代入したり、特定のメンバからデータを取り出したり、演算したりできます。つまり、構造体型変数の中の1つのメンバに対する操作は、通常の変数とまったく同様なのです。これはちょうど、配列の場合、添え字を使って1つの配列要素を指定すれば、変数と同様に扱えるのと同じことです。

【プログラム例】

```
/***************************************************
        構造体メンバ演算子
***************************************************/
#include <stdio.h>

/*構造体の宣言*/
typedef struct kojin_data{
        char    name[10];       /*氏名*/
```

次ページに続く

```
        int     age;                /*年齢*/
        double  length;             /*身長*/
}KOJIN;

main()
{
        KOJIN   student1 ={"kobayasi",12,155.3};
        int     old_length;


        /*1年間の成長*/
        student1.age ++;
        old_length = student1.length;               /*去年の身長保存*/
        student1.length = 160.4;                    /*今年の身長*/


        /*表示*/
        printf("氏名は%s¥n",student1.name);
        printf("年齢は%5d才¥n",student1.age);
        printf("身長は%7.1fcm¥n",student1.length);
        printf("去年より%5.1fcm大きくなりました¥n",
                student1.length - old_length);
```

メンバのデータに演算する

メンバを指定してデータを取り出す

メンバに代入する

配列型のメンバの表示

int型のメンバの表示

double型のメンバの表示

メンバと変数との演算

【実行結果】

```
氏名はkobayashi
年齢は    13才
身長は  160.4cm
去年より  5.4cm大きくなりました
```

●構造体のデータをまとめて代入

代入演算子=は、構造体型変数のすべてのデータを一度に複写します。

【プログラム例】

```
/************************************************
        構造体の代入
************************************************/
#include <stdio.h>

/*構造体の宣言*/
typedef struct kojin_data{
        char    name[10];          /*氏名*/
        int     age;               /*年齢*/
        double  length;            /*身長*/
}KOJIN;

main()
{
        /*変数の宣言*/
        KOJIN   student1 ={"kobayasi",12,155.3};
        KOJIN   student2;

        /*代入*/
        student2 = student1;   ← 丸ごと代入される

        /*表示*/
        printf("氏名は%s¥n",student2.name);
        printf("年齢は%5d才¥n",student2.age);
        printf("身長は%7.1fcm¥n",student2.length);
}                                  ↑ 新しい変数名を指定
```

【実行結果】

```
氏名はkobayasi
年齢は   12才
身長は  155.3cm
```

← student1のデータはすべてstudent2にコピーされている

●構造体型変数を指すポインタ

ポインタの指す先を構造体型変数にすることができます。構造体型といえども変数ですから、そのアドレスを得るには、アドレス演算子「&」を使います。また、ポインタは「構造体型を指すポインタ」として宣言します。

【プログラム例】

```
/*************************************************
        構造体を指すポインタ
*************************************************/
#include <stdio.h>

/*構造体の宣言*/
typedef struct kojin_data{
        char    name[10];           /*氏名*/
        int     age;                /*年齢*/
        double  length;             /*身長*/
}KOJIN;

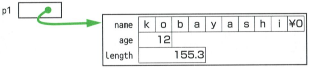
main()
{
        /*変数の宣言*/
        KOJIN   student1 ={"kobayasi",12,155.3};
        KOJIN   *p1;                /*student1を指すポインタ*/
        int     x=10;
        int     *px;                /* xを指すポインタ*/

        /*ポインタをつなぐ*/
        p1 = &student1;

        px = &x;
}
```

KOJIN型を指すポインタ

int型変数のときと、まったく同じ



●ポインタを使ったメンバの指定

ポインタが構造体型変数を指すとき、このポインタが指す先の構造体型変数を構成する1つのメンバを指定するときは、演算子「->」を使います。ポインタの間接演算子「*」や構造体メンバ演算子「.」は使いません。

【プログラム例】

```
/***********************************************
        ポインタを使ったメンバの指定
***********************************************/
#include <stdio.h>

/*構造体の宣言*/
typedef struct kojin_data{
        char    name[10];          /*氏名*/
        int     age;               /*年齢*/
        double  length;            /*身長*/
}KOJIN;

main()
{
        /*変数の宣言*/
        KOJIN   student1 ={"kobayashi",12,155.3};
        KOJIN   *p1;               /*student1を指すポインタ*/

        /*ポインタをつなぐ*/
        p1 = &student1;

        /*表示*/
        printf("氏名は%s¥n",p1->name);
        printf("年齢は%5d才¥n",p1->age);
        printf("身長は%7.1fcm¥n",p1->length);
 }
```

ポインタp1の指す先の
メンバ名nameの内容を表示

(注意) 演算子「->」は、半角のマイナス (-) と不等号 (>) とを組み合わせて、2文字で1つの演算子を表す記号です。全角の「→」ではありません。コンパイラは全角は認識できませんので、ご注意ください。

【実行結果】

```
氏名はkobayashi
年齢は　12才
身長は　155.3cm
```



●変数名とメンバ名

メンバ名は、構造体の型を決定する際、メンバ同士を識別するためにつける名前です。変数名とは使い方が異なりますから、同名のメンバ名と変数名が存在することが許されます。

【プログラム例】

```
/*************************************************
          メンバ名と変数名
*************************************************/
/*構造体の宣言*/
typedef struct kojin_data{
          char    name[10];        /*氏名*/
          int     age;             /*年齢*/
          double  length;          /*身長*/
}KOJIN;

main()
{
          KOJIN   student1 ={"kobayasi",12,155.3};
          char    name[10];
          int     age;
          double  length;
}
```

（図中の吹き出し：「メンバ名」＝構造体宣言中の length を指す。「変数名」＝main() 内の age を指す。）

6-3 構造体型配列

同じ型の構造体型変数をたくさん並べて構造体型配列を作ることができます。構造体の1個分が配列の1つの要素となります。型が構造体型であっても、通常の配列と同様ですから、配列名は、先頭の要素のアドレスを表します。また、添え字を使って、各要素を指定します。

●構造体型配列の定義

構造体型配列の定義は、通常の型の配列の定義とまったく同様です。

```
struct 構造体タグ　配列名[要素数] = {初期化子の並び};
```

または

```
型名　　配列名[要素数] = {初期化子の並び};
```

【プログラム例】

```
/************************************************
        構造体型配列
************************************************/
/*構造体の宣言*/
typedef struct kojin_data{
        char    name[10];           /*氏名*/
        int     age;                /*年齢*/
        double  length;             /*身長*/
}KOJIN;

main()
{
        /*配列の宣言*/
        KOJIN   student[3] = {  {"kobayashi",12,155.3},
                                {"ariyoshi" ,11,148.2},
                                {"nakaura"  ,12,154.8}  };
}
```

student[0]	name	k	o	b	a	y	a	s	h	i	¥0
	age	12									
	length	155.3									
student[1]	name	a	r	i	y	o	s	h	i	¥0	
	age	11									
	length	148.2									
student[2]	name	n	a	k	a	u	r	a	¥0		
	age	12									
	length	154.8									

●構造体型配列のデータの表示

構造体型配列では、添え字を使って配列要素を指定し、さらに、構造体メンバ演算子を使ってメンバを指定します。

【プログラム例】

```
/**************************************************
        構造体型配列の表示
**************************************************/
#include <stdio.h>

/*構造体の宣言*/
typedef struct kojin_data{
        char    name[10];        /*氏名*/
        int     age;             /*年齢*/
        double  length;          /*身長*/
}KOJIN;

main()
{
        /*配列の宣言*/
        KOJIN   student[3] = { {"kobayashi",12,155.3},
                               {"ariyoshi",11,148.2},
                               {"nakaura",12,154.8}  };
        int     i;      /*添え字*/
```

次ページに続く

```
	/*データの表示*/
	printf("メンバー紹介¥n");
	printf("   氏名        年齢  身長¥n");
	for(i=0;i<3;i++){
		printf("%-10s  %3d  %7.1f¥n",
			student[i].name,
			student[i].age,
			student[i].length);
	}
}
```

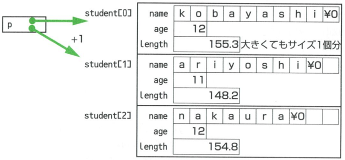
配列studentのi番目の要素の、指定メンバに関するデータを表示

【実行結果】

```
メンバー紹介
   氏名        年齢  身長
kobayashi     12   155.3
ariyoshi      11   148.2
nakaura       12   154.8
```

●構造体型配列を指すポインタ

配列名が先頭の要素のアドレスを持つことは、構造体型であっても同じです。また、構造体型を指すポインタの値に1を加えると、要素1個分のサイズが加算され、配列の次の要素を指すようになることも、通常の配列と同様です。



【プログラム例】

```
/***********************************************
        構造体型配列の表示
***********************************************/
#include <stdio.h>

/*構造体の宣言*/
typedef struct kojin_data{
        char    name[10];               /*氏名*/
        int     age;            /*年齢*/
        double  length;         /*身長*/
}KOJIN;

main()
{
        /*配列の宣言*/
        KOJIN   student[3] = {  {"kobayashi",12,155.3},
                                {"ariyoshi",11,148.2},
                                {"nakaura",12,154.8}  };

        KOJIN   *p;             /*student を指すポインタ*/
        int     i;              /*カウンタ*/

        /*ポインタをつなぐ*/
        p = student;

        /*データの表示*/
        printf("メンバー紹介¥n");
        printf("  氏名     年齢  身長¥n");
        for(i=0;i<3;i++,p++){

                printf("%-10s  %3d  %7.1f¥n",
                            p->name , p->age , p->length);

        }
}
```

3人分繰り返すのを数える

配列なので、&は不要

p++で次の人を指す

6-4 構造体型のメンバ

構造体では、int型やchar型などの通常の変数の他に、配列・ポインタ・他の型の構造体など、あらゆる型をメンバにすることができます。他の型の構造体をメンバにすると、1件分のデータ群を小さなグループに分けるなど、意味付けすることができます。

●別の型の構造体をメンバにする

あらかじめ宣言されている別の型の構造体をメンバにすることができます。そのような構造体で内側のメンバの1つを指定するには、構造体メンバ演算子を2つ使います。

【プログラム例】

```
/**********************************************
          構造体型配列の表示
**********************************************/
#include <stdio.h>

/*構造体の宣言*/
typedef struct kojin_data{
        char    name[10];         /*氏名*/
        int     age;              /*年齢*/
        double  length;           /*身長*/
}KOJIN;

typedef struct team_member{
        int     uniformNo;        /*背番号*/
        KOJIN   info;             /*個人情報*/
        char    pos[3];           /*ポジション*/
}PLAYER;
```

次ページに続く

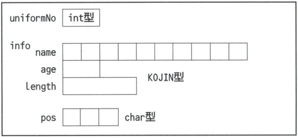
```
main()
{
        /*配列の宣言*/
        PLAYER   team1[3] = {{1,{"kobayashi",12,155.3},"GK"},
                            {2,{"ariyoshi",11,148.2},"FW"},
                            {3,{"nakaura",12,154.8},"MF"} };

        int     i;                  /*添え字*/

        /*データの表示*/
        printf("メンバー紹介¥n");
        printf("背番号 氏名         年齢  身長  ポジション¥n");
        for(i=0;i<3;i++){
              printf("  %2d   %-10s  %3d %7.1f    %s¥n",
                     team1[i].uniformNo,
                     team1[i].info.name,team1[i].info.age,
                     team1[i].info.length,team1[i].pos);
        }
}
```

配列名[添え字].メンバ名.メンバ名

構造体型名PLAYER



【実行結果】

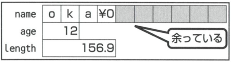
```
メンバー紹介
背番号 氏名        年齢  身長  ポジション
  1   kobayashi   12   155.3   GK
  2   ariyoshi    11   148.2   FW
  3   nakaura     12   154.8   MF
```

6-5 ポインタ型のメンバ

データの長さが可変であったり、あらかじめサイズを予測することができない場合には、実際のデータを記録する領域を構造体の外側に確保し、そこを指すポインタだけを構造体メンバにすることで、柔軟性の高いプログラムにすることができます。

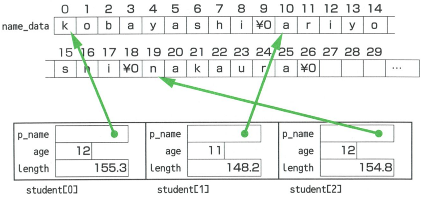
● char型を指すポインタをメンバにする

たとえば、構造体に文字列を記録したいとき、構造体内にchar型配列を確保すると、最も長い文字列に合わせてchar型配列のサイズを決めなければなりません。短い文字列を記録したい場合には、メモリ領域が余ってしまいます。また、宣言したサイズ以上の長さの文字列は記録することができません。





そこで、構造体とは別に長いchar型配列を確保して、実際のデータはこの配列にどんどん詰めます。char型を指すポインタ型の構造体メンバが配列内の該当するデータを指すようにすることで、メモリを無駄なく使い、かつ、どんな長さの文字列にも対応できる柔軟なプログラムにすることができます。



構造体型配列の各要素では、ポインタ型メンバ**p_name**が、それぞれ自分の該当するデータを指しています。

【プログラム例】

```
/************************************************
        ポインタメンバ
************************************************/
#include <stdio.h>
#include <string.h>

/*構造体の宣言*/
typedef struct kojin_data{
        char    *p_name;            /*氏名*/
        int     age;                /*年齢*/
        double  length;             /*身長*/
}KOJIN;

main()
{
        /*配列の宣言*/
        KOJIN   student[3] = {{NULL,12,155.3},
                              {NULL,11,148.2},
                              {NULL,12,154.8} };
        char    name[3][10]={"kobayashi","ariyoshi","nakaura"};
```

次ページに続く

```
    char    name_data[100]; /* 氏名の情報をすべて記録*/
    char    *p;             /* name_dataを指すポインタ*/
    int     i;              /*添え字*/

    /*名前データの記録*/
    p = name_data;          /* pがname_dataの先頭を指す*/

    for(i=0;i<3;i++){
            student[i].p_name = p;  /*ポインタを設定*/
            strcpy(p,name[i]);      /*生徒の氏名を記録*/
            p = p + strlen(p)+1;    /*文字数分先に進める*/
    }

    /*データの表示*/
    printf("メンバー紹介¥n");
    printf("氏名        年齢  身長  ¥n");
    for(i=0;i<3;i++){
        printf("%-10s  %3d %7.1f¥n",
                student[i].p_name,
                student[i].age,
                student[i].length);
    }
}
```

【実行結果】

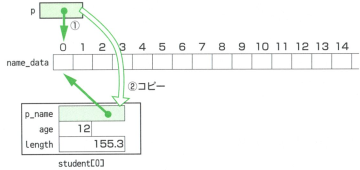
```
メンバー紹介
氏名        年齢  身長
kobayashi    12   155.3
ariyoshi     11   148.2
nakaura      12   154.8
```

●ポインタをつなぐ様子

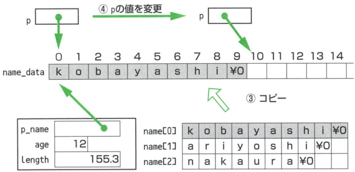
前ページのプログラムでは、氏名の文字列は別の配列で与えており、これを、次々に長いchar型配列name_dataにコピーしながら、構造体メンバp_nameが該当する氏名の先頭のアドレスを指すように、データを構築しています。その様子を下図で説明します。

pは、これから文字列を記録しようとする位置を指すポインタです。初めは

name_dataには何もデータが入っていませんので、pはname_dataの先頭を指すようにします（①）。



先頭のデータ（student[0]）のp_nameも、同じアドレスを指すことになりますので、pの内容をstudent[0].p_nameにコピーします（②）。pの指す先には、配列nameで指定された文字列をコピーします（③）。



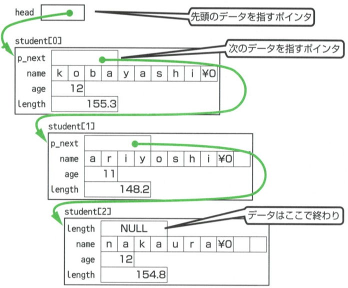
最後にpの値を、次のデータをコピーする位置に変更します（④）。
以上を、データがあるだけ繰り返すのです。

6-6 自分と同じ構造体を指すポインタ型メンバ

構造体型メンバに自分と同じ構造体型を指すポインタがあると、バラバラのデータを順にポインタでつないで、一元的に管理することができます。このようなデータ構造をリストの鎖状配置といいます。

●自分と同じ構造体型を指すポインタをメンバにする

自分と同じ構造体型を指すポインタを構造体メンバにすると、同じ型のデータを次々とポインタでつなぐデータ構造を作ることができます。このとき、先頭のデータを指すためのポインタを設けておきます。



このデータ構造を実現するためには、次のような構造体宣言を行います。

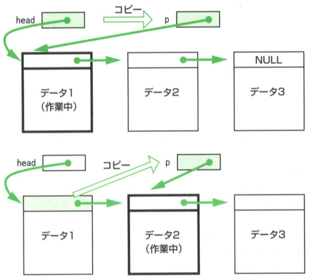
```
/*構造体の宣言*/
typedef struct kojin_data1{
        struct kojin_data1     *p_next;         /*次へのポインタ*/
        char                   name[10];        /*氏名*/
        int                    age;             /*年齢*/
        double                 length;          /*身長*/
}KOJIN1;
```

自分と同じ構造体型を指すポインタ

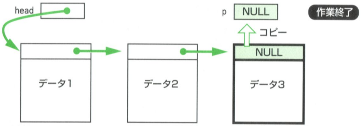
このように、同じ構造のデータを次々と指すデータ構造を、**リストの鎖状配置**といいます。

●リストの鎖状配置のデータ表示

鎖状配置されているデータを順に表示します。構造体を指すポインタをもう1つ用意し、作業（表示）しているデータを指すようにします。作業（表示）が終わると、次を指すようにポインタの値を更新します。上の例では、p_nextの内容をpにコピーすることになります。



最後にデータ3のp_nextをpにコピーすると、pがNULLとなり、作業は終了します。



【プログラム例】

```
/************************************************
        自分と同じ型を指すポインタメンバ
************************************************/
#include <stdio.h>

/*構造体の宣言*/
typedef struct kojin_data1{
        struct kojin_data1      *p_next;        /*次へのポインタ*/
        char                    name[10];       /*氏名*/
        int                     age;            /*年齢*/
        double                  length;         /*身長*/
}KOJIN1;

main()
{
        /*配列の宣言*/
        KOJIN1  student1 = {NULL,"kobayashi",12,155.3};
        KOJIN1 student2 = {NULL,"ariyoshi" ,11,148.2};
        KOJIN1 student3 = {NULL,"nakaura"  ,12,154.8};
        KOJIN1 *head;           /*ヘッダ*/
        KOJIN1 *p;              /*データを指すポインタ*/

        /*ポインタをつなぐ*/
        head = &student1;
        student1.p_next = &student2;
```

次ページに続く

```
        student2.p_next = &student3;

        /*データの表示*/
        printf("メンバー紹介¥n");
        printf("氏名          年齢  身長  ¥n");
        for(p = head ; p != NULL ; p = p->p_next){
                printf("%-10s  %3d %7.1f¥n",
                        p->name,p->age,p->length);
        }
}
```

【実行結果】

```
メンバー紹介
氏名        年齢  身長
kobayashi    12   155.3
ariyoshi     11   148.2
nakaura      12   154.8
```

構造体型配列ではデータの順が決まってしまうのに対し、ポインタでつなぐリストの鎖状配置は、ポインタの内容さえ変えれば、容易にデータの順序を変更したりデータを追加・削除することができます

6-7 ファイル操作とは

データをプログラムとは別のファイルに保存することで、プログラムとデータとを切り分けることができます。また、同じデータに対し、別のプログラムで別の処理をすることもできますし、処理した結果を保存しておいて何度も参照することもできます。

●ファイル操作とは

コンピュータ処理の基本は、入力されたデータをコンピュータによって加工し、これを出力することです。今まで、入力はキーボードから、出力は画面に表示してきました。

▼キーボード入力・画面入力



最初の一度は、なんらかの方法で人間が入力をしなければなりませんが、それをファイルに記録しておけば、そのデータを何度でも使うことができます。ファイルとのデータのやりとりを**ファイル操作**といいます。

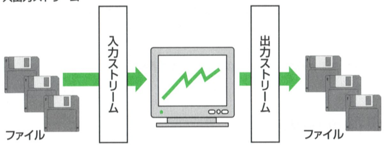
▼ファイル操作



●ファイルとコンピュータをつなぐ「道」ストリーム

ハードディスクやフロッピーディスクなどに保存されているファイルを操作するためには、ハードウェア（物理装置）が必要ですが、ハードウェアとファイルとをつなぐ論理的な領域を**ストリーム**といいます。ストリームは、ファイル内のデータとコンピュータとの間の「道」のようなもので、一時的にデータが保存されるバッファの役目も果たします。大きな道路には、上り車線と下り車線とがあるように、ストリームも、コンピュータへの上り車線としての入力ストリームと、コンピュータからの下り車線としての出力ストリームとを使います。また、ストリームの種類として、テキストデータを扱うテキストストリームと、バイナリファイルを扱うバイナリストリームとが用意されています。

▼入出力ストリーム



ファイルは、入出力に先立って、オープンします。オープンすることで、ファイルとストリームが結び付けられます。また、クローズすることで、ファイルはストリームから切り離されます。

今まで使ってきたキーボードからの入力や画面への表示も、同様なストリームを介しています。これらは、標準入出力といって、プログラム開始時に自動的にオープンされ、プログラムが終了すると自動的にクローズされるので、いきなり`printf`で画面に表示したり、`scanf`でキーボードから入力したりすることができたのです。

6-8 ファイル処理を行うライブラリ関数

ファイル操作は、決まった手順で行います。その手順の1つ1つがライブラリ関数として用意されていますので、関数の使い方を理解してください。また、ファイルが存在しなかった、読めなかった、などエラーが発生した場合の処理が不可欠となります。

●ストリームを制御する構造体FILE

ストリームを制御するには、stdio.hに定義されている構造体FILE型の変数を使います。この変数は、ファイルをオープンしたときに作られます。ですから、ソースプログラム上では、この変数を指すポインタだけを確保します。

```
FILE  *fp;     /*FILE型変数へのポインタの宣言*/
```

●ファイルのオープン

ファイルのオープンは、ライブラリ関数fopenを使います。どのファイルをオープンするのか（ファイル名）のほかに、オープンに伴って結び付けるストリームの種類も指定します。

[ヘッダ]　stdio.h

[形　式]　FILE *fopen(const char *filename,const char *mode);

[引　数]　filename：ファイル名を表す文字列を指すポインタ

mode：結びつけるストリームの種類を表す文字列

（右表のいずれかを指定します）

[返却値]　ファイルのオープンが成功したときは、作成されたFILE型変数へのポインタ。失敗したときは、NULL。

▼ modeに指定する主な文字列

文字列	ストリームの種類	備考
"r"	テキスト入力	ファイルが存在しない・読めないときエラー
"w"	テキスト出力	ファイルがすでに存在しているときは内容をすべて消去し、存在しないときは新たに作る
"a"	テキスト出力	追加書き込みを行う。ファイルが存在しないときは新たに作る
"rb"	バイナリ入力	ファイルが存在しない・読めないときエラー
"wb"	バイナリ出力	ファイルがすでに存在しているときは内容をすべて消去し、存在しないときは新たに作る
"ab"	バイナリ出力	追加書き込みを行う。ファイルが存在しないときは新たに作る

●ファイルのクローズ

ファイルをオープンしたときに得たFILE型変数へのポインタを使って、ファイルをクローズし、ストリームを切り離します。

[ヘッダ]　stdio.h

[形　式]　int fclose(FILE *stream);

[引　数]　stream：そのファイルをオープンしたときに得たポインタ

[返却値]　成功したときは0　失敗したときはEOF



ファイル操作の手順は

1. ファイルを開く
2. 読み書きなど
3. ファイルを閉じる

となり、通常の紙ファイルと同様です。

【プログラム例】

ファイル名indata.txtをテキスト入力モードでオープンします。メモ帳などのテキストエディタでindata.txtという名前でテキストファイルを作成し、あらかじめプログラムと同じフォルダに入れてください。

```
#include <stdio.h>
#include <stdlib.h>

main()
{
        FILE *fpi;

        /*入力ファイルのオープン*/
        if((fpi = fopen("indata.txt","r")) == NULL)
        {
            printf("can't open --- indata.txt¥n");
            exit(1);
        }
        :
        :
        /*ファイルのクローズ*/
        fclose(fpi);
}
```

ライブラリ関数exit()のヘッダ

エラーが発生し、これ以上処理を続けられないとき、プログラムを終了する。通常、正常終了のときは exit(0); とするが、それ以外の実引数が与えられたときは、エラー終了であることをMS-DOSに知らせる

●ファイルからのテキスト入力

ファイルから文字を読み込むライブラリ関数はいくつかありますが、ここでは、1行分を一度に入力する関数を紹介します。

［ヘッダ］ stdio.h

［形　式］ char *fgets(char *s , int n , FILE *stream);

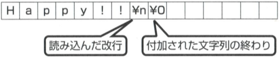
［引　数］ s：入力した文字列を記録するchar型配列へのポインタ

n：入力できる最大の文字数

stream：そのファイルをオープンしたときに得たポインタ

[返却値] ファイルからの入力が成功したときは、文字列へのポインタsを返す。入力に失敗したとき、1文字も入力せずにファイルの終わり(EOF)を検出したときはNULLを返す。
改行またはEOFに出会うか、n-1文字を読み込むまで1文字ずつ入力を続け、最後に'¥0'をつける。読み込んだ改行も配列に入る。

▼1行入力したときのchar型配列の内容の例



【プログラム例】

テキストファイルindata.txtから文字列を読み込んで、画面に表示するプログラムです。indata.txtは、あらかじめプログラムと同じフォルダに入れてください。

```
/*************************************
        ファイルからの入力
*************************************/
#include <stdio.h>
#include <stdlib.h>

main()
{
        FILE    *fpi;               /*入力ファイル*/
        char    buff[80];           /*作業用バッファ*/

        /*入力ファイルのオープン*/
        if((fpi = fopen("indata.txt","r")) == NULL)
        {
                printf("can't open --- indata.txt¥n");
                exit(1);
        }
```

次ページに続く

入力した文字列を

```
        /*ファイルからの入力*/
        while(fgets(buff,80,fpi) != NULL)
        {
                printf("%s",buff);          /*入力データを表示*/
        }

        /*ファイルのクローズ*/
        fclose(fpi);
}
```

画面に表示

●ファイルに対するテキスト出力

ファイルに文字を書き出すライブラリ関数はいくつかありますが、ここでは、1行分を一度に出力する関数を紹介します。

[ヘッダ]　stdio.h

[形　式]　int　fputs(const char　*s , FILE *stream);

[引　数]　s：出力したい文字列が記録されているchar型配列へのポインタ
　　　　　stream：そのファイルをオープンしたときに得たポインタ

[返却値]　ファイルへの出力が成功したときは、負でない値を返す。失敗したときは、ファイルの終わりを表す記号（EOF）を返す。

【プログラム例】

テキストファイルindata.txtから文字列を読み込んで、テキストファイルoutdata.txtに出力する、つまりファイルの内容を別のファイルにコピーするプログラムです。indata.txtは、あらかじめプログラムと同じフォルダに入れてください。

```
#include <stdio.h>
#include <stdlib.h>
```

次ページに続く

```
main()
{
        FILE    *fpi;           /*入力ファイル*/
        FILE    *fpo;           /*出力ファイル*/
        char    buff[80];       /*作業用バッファ*/

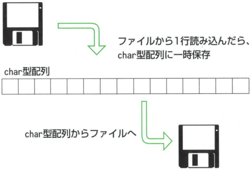
        /*入力ファイルのオープン*/
        if((fpi = fopen("indata.txt","r")) == NULL)
        {
                printf("can't open --- indata.txt¥n");
                exit(1);

        }

        /*出力ファイルのオープン*/
        if((fpo = fopen("outdata.txt","w")) == NULL)
        {
                printf("can't open --- outdata.txt¥n");
                exit(1);
        }

        /*ファイルのコピー*/
        while(fgets(buff,80,fpi) != NULL)
        {
                fputs(buff,fpo);
        }

        /*ファイルのクローズ*/
        fclose(fpi);
        fclose(fpo);
}
```

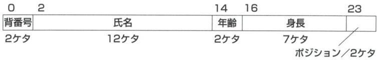


6-9 ファイルの情報を構造体に読み込む

1件分のデータとして、何種類かの情報があるとき、これをテキストファイルの1行に記録しておきます。このような情報のまとまりを1レコードといいます。ファイルはレコードの集まりです。これらの情報はレコードごとに構造体型の変数に読み込んで処理します。

●ファイルの形式を決める

まず、1件分のデータとして、どんなデータを何桁目から記録するかを決定します。これがテキストファイルの1行分になります。



ファイルの1番先頭には記録されているデータの件数（レコードの数）を入れておくことにしましょう。このようなファイルの形式を決める作業を、**ファイルの設計**といいます。右図は、上の約束に従って作ったデータファイルをメモ帳で表示したところです。

▼データファイル例



```
15
 1小林    敦  12 155.3 GK
 2有吉  隼也  11 148.2 DF
 3中浦  祥太  12 154.8 DF
 4松野下裕基  12 156.4 DF
 5坪井    渉  11 144.5 MF
 6板倉  隆行  11 150.1 DF
 7大野竜太郎  11 150.9 MF
 8福田  利貞  12 152.0 MF
 9岡    正道  12 153.6 FW
10小田  淳士  12 159.8 FW
11木川  俊成  12 151.7 FW
12高田  駿士  11 149.9 MF
13前田  大介  11 150.2 DF
14森田  祐規  12 146.3 FW
15熊谷  達也  12 159.0 MF
```

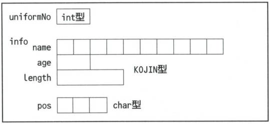

先頭の「15」はデータの件数。このファイルには15件のデータが記録されています

●構造体の型を決める

1件分のデータを読み込むための構造体の型を決めます。ファイルには、す

べての情報が文字で記録されていますが、構造体メンバは適切な型で宣言します。文字型でない情報は、読み込む過程で型変換をします。ここでは、6-4で決めた型を使うことにしましょう（注意：データの特性により、氏名を記録するchar型配列nameの要素数は変更しています）。

構造体型名PLAYER



●データを表示するプログラム

データが記録されているファイルmember.txtの内容を構造体型配列に読み込んで、画面に表示するプログラムです。記録されているデータの件数は、ファイルを開いて先頭の1行を読み込んでみないとわからないので、構造体型配列のためのメモリ領域をあらかじめ確保することができません。ライブラリ関数mallocを使って必要なだけ確保することにします。

文字列データである「氏名」と「ポジション」は、文字列を指定文字数だけ複写するライブラリ関数strncpyを使って構造体メンバに代入します。年齢・身長は、数値を文字列に変換する関数atoi、atofを使います。プログラムは少し長めになりますが、今までの総まとめです。

なお、簡単にするために、エラーチェックとエラーの場合の処理を若干省略してあります。

【プログラム例】

```
/**********************************
       ファイル操作と構造体
**********************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*構造体の宣言*/
typedef struct kojin_data{
        char    name[14];       /*氏名*/
        int     age;            /*年齢*/
        double  length;         /*身長*/
}KOJIN;

typedef struct team_member{
        int     uniformNo;      /*背番号*/
        KOJIN   info;           /*個人情報*/
        char    pos[3];         /*ポジション*/
}PLAYER;

main()
{
        FILE    *fpi;           /*入力ファイル*/
        char    buff[80];       /*作業用バッファ*/
        int     n;              /*データの件数*/
        PLAYER  *p;             /*構造体を指すポインタ*/
        int     i;              /*添え字*/

        /*入力ファイルのオープン*/
        if((fpi = fopen("member.txt","r")) == NULL)
        {
                printf("can't open --- indata.txt¥n");
                exit(1);
        }

        /*件数の読み込み*/
        fgets(buff,80,fpi);     /*先頭1行読み込む*/
        n = atoi(buff);         /*int型に変換*/
```

構造体型をメンバに含みます

変数nに件数が求まります

次ページに続く

```
/*メモリの確保*/
p = malloc(sizeof(PLAYER)*n);

/*ファイルからの入力*/
for(i=0;i<n;i++)
{
        fgets(buff,80,fpi);        /*1件読み込み*/

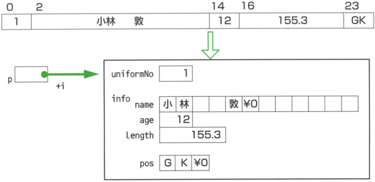
        /*構造体に記録*/
        (p+i)->uniformNo = atoi(buff);
        strncpy((p+i)->info.name , &buff[2],12);  /*氏名*/
        (p+i)->info.name[12]='¥0';
        (p+i)->info.age = atoi(&buff[14]);          /*年齢*/
        (p+i)->info.length = atof(&buff[16]);       /*身長*/
        strncpy((p+i)->pos , &buff[23],2);          /*ポジション*/
        (p+i)->pos[2]='¥0';
}
```

本来なら、メモリを確保できなかったときのエラー処理が必要

構造体1個分のサイズをsizeof演算子で求め、それが件数分必要

下に続く

1件分のデータは、下図のように構造体に読み込まれます。




```
/*ファイルのクローズ*/
fclose(fpi);

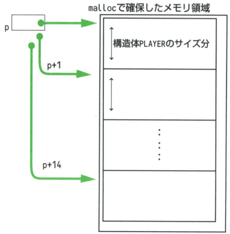
/*表示*/
printf("メンバー紹介¥n");
```

読み込みが完了してしまえば、もうファイルは不要

次ページに続く

```
    printf("背番号 氏名          年齢  身長  ポジション¥n");
    for(i=0;i<n;i++){
        printf("  %2d   %-10s  %3d %7.1f    %s¥n",
               (p+i)->uniformNo,(p+i)->info.name,
               (p+i)->info.age,(p+i)->info.length,
               (p+i)->pos);
    }

    /*メモリの開放*/
    free(p);
}
```



【実行結果】

```
メンバー紹介
背番号 氏名         年齢  身長  ポジション
   1   小林    敦    12   155.3    GK
   2   有吉  隼也    11   148.2    DF
   3   中浦  祥太    12   154.8    DF
   4   松野下裕基    12   156.4    DF
   5   坪井    渉    11   144.5    MF
   6   板倉  隆行    11   150.1    DF
   7   大野竜太郎    11   150.9    MF
   8   福田  利貞    12   152.0    MF
   9   岡    正道    12   153.6    FW
  10   小田  淳士    12   159.8    FW
  11   木川  俊成    12   151.7    FW
  12   高田  駿士    11   149.9    MF
  13   前田  大介    11   150.2    DF
  14   森田  祐規    12   146.3    FW
  15   熊谷  達也    12   159.0    MF
```

このプログラムでは、ファイルから読み込んだデータをそのまま表示しただけですが、たとえば、背の高い順に並べ替えて表示したり、ポジションが「FW」のデータだけを抽出して別のファイルに出力したり、といったデータの加工を施す処理を**データ処理**といいます。

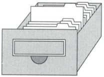
COLUMN

データ構造

コンピュータによる仕事の多くは、データを処理することです。実際の業務では、大量のデータを扱うことになります。この大量のデータは、ただ保存しているだけではなく、必要なものを必要なときにすばやく探し出してこそ、意義のあるものとなります。データ処理とは、データの検索・加工・そして、再び保存すること、といえるでしょう。

ここで、大量の名刺の中から、ある1枚を探し出したいとしましょう。名刺がバラバラに入っていたのでは、すべての名刺を1枚1枚調べなければなりません。ですが、名刺が氏名の昇順に並べられていれば、国語辞典を引く要領ですばやく探し当てることができます。一方で、常に昇順に整理しておかなければならないのは、逆に相当骨の折れる作業です。

それならば、「あ行」の名刺、「か行」の名刺、「さ行」の名刺くらいには分けておいて、その中ではランダムに並べてよいようにしておいてはどうでしょう? 探す範囲が狭くてすむ上に、普段の整理もそれほどおっくうにはならないでしょう。



このように、そのシステムにとって、データをどのように整理して保存しておくのがトータル的に最も効率がよいかを考えることは、非常に重要な作業なのです。データの保存のしかたをデータ構造といい、代表的なリストの鎖状配置のほか、多数のデータ構造が考案されています。

COLUMN

INDEX

索引

記号・数字

A～G

I～W

あ行

ま～ら 行

■著者紹介

■高田美樹（たかたみき）

慶応義塾大学工学部電気工学科卒。
大手電気メーカーにてマイクロコンピュータのソフトウェア開発に従事。
その後、SOHOという言葉もまだない時代から在宅でソフトを開発、そのかたわら専門学校の講師を勤める。現在、横浜電算学院講師。
主な著書に、『基本情報技術者試験 らくらく突破C言語』『C言語スタートブック』（共に技術評論社）がある。

カバー・本文デザイン ◆ Kuwa Design＋桑原文子
カバーイラスト ◆ 吉田雅博
本文イラスト ◆ Handsome Inc.
本文イラスト・DTP ◆ メグテック

まるごと図解
最新C言語がわかる

平成14年10月11日　初版　第1刷発行

著　者　髙田　美樹
発行者　片岡　巖
発行所　株式会社技術評論社
東京都品川区上大崎3-1-1
電話　03-5745-7800　販売促進部
03-5745-7830　書籍編集部
印刷／製本　日経印刷株式会社

定価は表紙に表示してあります。

ISBN4-7741-1572-X C3055
Printed in Japan

本書の内容に関するご質問は下記の宛先までFAXか書面でお送りください。電話によるご質問には一切お答えできません。また、本書に記述されている内容以外のご質問にはお答えできませんのでご了承ください。

〒141-8676
株式会社技術評論社　書籍編集部
「まるごと図解　最新C言語がわかる」質問係
FAX：03-5745-7831

ISBN4-7741-1572-X C3055 ¥1580E

定価(本体1580円十税) M-code 404331

9784774115726

1923055015807

まるごと図解

最新C言語がわかる